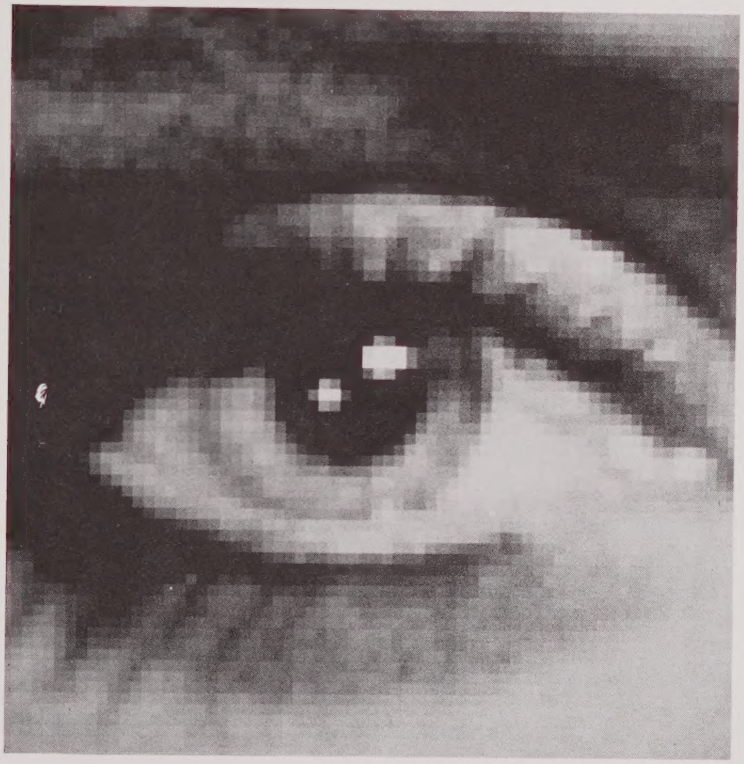
COGNEX

Vision for Industry

COGNEX

Vision For Industry

Cognex 2000/3000/4000/5000
Programming Reference
Revision 6.0
P/N 590-0104



Cognex 2000/3000/4000/5000
Programming Reference
Revision 6.0
P/N 590-0104

Cognex 2000/3000/4000/5000
Programming Reference
Revision 6.0

The software described in this document is furnished under license, and may be used or copied only in accordance with the terms of such license and with the inclusion of the copyright notice shown on this page. Neither the software, this document, nor any copies thereof may be provided to or otherwise made available to anyone other than the licensee. Title to and ownership of this software remains with Cognex Corporation or its licensor.

Cognex Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Cognex Corporation. Cognex Corporation makes no warranties, either express or implied, regarding the described software, its merchantability or its fitness for any particular purpose.

The information in this document is subject to change without notice and should not be construed as a commitment by Cognex Corporation. Cognex Corporation is not responsible for any errors that may be present in either this document or the associated software.

Copyright©1993
Cognex Corporation
All Rights Reserved
Printed in U.S.A.

This document may not be copied in whole or in part, nor transferred to any other media or language, without the written permission of Cognex Corporation.

The following are trademarks of Cognex Corporation:

COGNEX logo	COGNEX 2000	COGNEX 5000
COGNEX 1500	COGNEX 3000	VC-1
COGNEX 1600	COGNEX 4000	VC-2

The following are registered trademarks of Cognex Corporation:

DataMan	CHECKPOINT	COGNEX, Vision for Industry
On-Sight		

The following are trademarks or registered trademarks of the associated company:

UNIX: AT&T

Epsilon, Lugaru: Lugaru Software, Ltd.

Microtec: Microtec Research, Inc.

MS-DOS: Microsoft Corp.

IBM: International Business Machines Corp.

Sun, Sun-2, Sun-3, Sun-4, Sun Workstation: Sun Microsystems, Inc.

VMEbus, Motorola, MC68000, MC68020, MC68030, MC68881, MC68882: Motorola, Inc.

DEC: Digital Equipment Corporation

Cognex Corporation serves its customers from the following locations:

Eastern U.S.:

15 Crawford Street
Needham, MA 02194
Phone: 617-449-6030
Fax: 617-449-4013

Western U.S.:

Regional Technology Center,
1949 Landings Drive
Mountain View, CA 94043
Phone: 415-969-4812
Fax: 415-969-4818

Japan:

Towa Komagome Building 6F
5-4-7 Honkomagome, Bunkyo-ku
Tokyo 113, Japan
Phone: 011-81-3-5395-4791
Fax: 011-81-3-5395-4695

Germany:

Weidenbornstraße 8A
6200 Wiesbaden
Germany
Phone: 011-49-611-7901118
FAX: 011-49-611-7901119

France:

131-135 Boulevard Carnot
78110 - Le Vesinet
France
Phone: 011-33-1-3071-42-42
Fax: 011-33-1-3053-26-21

BOX and I 1
c_function 2
caq_acquire() 3
caq_arm() 5
caq_async_end() 6
caq_async_init() 7
caq_blink_enable 8
caq_cancel() 9
caq_constant() 10
caq_context 11
caq_cpu() 19
caq_display() 20
caq_even 21
caq_even_line 22
caq_field 23
caq_format 24
caq_frame 26
caq_get_format_name() 27
caq_gplane 28
caq_graphics() 29
caq_graphics_buffers() 30
caq_graphics_color() 31
caq_graphics_select() 32
CAQ_GROUP() 33
caq_image 34
caq_image_buffers() 35
caq_image_select() 36
caq_init() 37
caq_init_moving() 39
caq_line 40
caq_load_vb_action() 41
caq_mode 42
caq_odd 43
caq_odd_line 44
caq_overlay() 45
caq_ready() 46
caq_realtime() 47
caq_reset_vb_action() 48
caq_resolution() 49
caq_rgplane 50
caq_set_gplane_enable() 51
caq_specify_general_format() 52
caq_startup_context 54
caq_stored_eightbit 56
caq_stored_field 57
caq_stored_image_polarity 58
caq_sync_out 59
caq_sync_src 60
caq_video() 61
caq_video_mode 62
cblob_connectivity() 63

cblob_one() 64
cblob_params 65
cblob_regions() 67
cblob_results 68
cblob_two() 72
cblob_zero() 73
CBUS_ASSERT_RESET 74
cbus_clear_isa_int() 75
cbus_clear_vme_int() 76
CBUS_DEASSERT_RESET 77
CBUS_EXP_INT() 78
CBUS_IPCR6 79
CBUS_MAGIC_ADDR 80
cbus_map_vme_addr() 81
CBUS_PTR_ADDR 83
cbus_register_interrupter() 84
cbus_regs 85
cbus_set_isa_int() 89
cbus_set_state() 90
cbus_set_vme_int() 91
CBUS_SHARED_BASE_ADDRESS 92
CBUS_SHARED_LENGTH 93
CBUS_SHARED_USER_BASE 94
CBUS_SHARED_USER_LENGTH 95
cbus_test_isa_int() 96
CBUS_TEST_SET() 97
cbus_test_vme_int() 98
CBUS_TRIGGER() 99
cbus_unmap_vme_addr() 100
CBUS_VER_ADDR 101
ccal_default_ago 102
ccal_scale() 103
ccal_scale_params 104
ccal_size_init() 105
ccal_size_run() 106
ccal_size_term() 107
ccal_video_params 108
ccal_video_setup() 110
cclp_apply() 113
cclp_caliper 114
cclp_cleanup() 117
cclp_constraint 118
cclp_edge 119
cclp_init() 120
cclp_params 121
cclp_results 123
ccm_read() 124
ccm_write() 125
ccol_buffer 126
ccol_copy() 129
ccol_create() 130

ccol_csys 131
ccol_delete() 132
ccol_delete_planes() 133
ccol_expand() 134
ccol_link() 136
ccol_overlap() 137
ccol_pt_xform() 138
ccol_sample() 139
ccol_set() 141
ccol_window() 142
ccol_xform() 143
cct_catch() 145
cct_end() 146
cct_error() 147
cct_name() 148
cct_signal 149
CCT_SIG() 150
CCT_SIG_E() 151
CCT_SIG_G() 152
cct_throw() 153
cct_willcatch() 154
cd_bkpt_id 155
cd_bkpt() 156
cd_bkpt_pred() 157
cd_clear_bkpt() 158
cd_clear_trace() 159
cd_dsc() 160
cd_dump() 161
cd_ftrace() 162
cd_full_debug 163
cd_maxlevel 164
cd_pstate 165
cd_R 166
cd_show() 167
cd_show_bkpt() 168
cd_show_pstate() 169
cd_show_trace() 170
cd_showas() 171
cd_strace() 172
cd_undefined() 173
cdb_check_sizes() 174
cdb_close() 175
cdb_context 176
cdb_copy_file() 177
cdb_copy_record() 178
cdb_dump_index() 179
CDB_E() 180
CDB_EK() 181
cdb_entry 182
cdb_exit() 185
cdb_file 186

cdb_free_record() 187
cdb_get_acqimg() 188
cdb_get_acqimg_data() 189
cdb_get_acqimg_hdr() 190
cdb_get_acquire() 191
cdb_get_command() 192
cdb_get_data() 193
cdb_get_file() 194
cdb_get_image() 195
cdb_get_image_data() 196
cdb_get_image_hdr() 197
cdb_get_record() 198
cdb_get_record_comment() 199
cdb_get_record_data() 201
cdb_get_record_hdr() 203
cdb_get_response() 204
cdb_get_result() 205
cdb_global_sequence 206
cdb_index 207
cdb_init() 208
cdb_key_open() 209
cdb_open() 211
cdb_put_acquire() 213
cdb_put_command() 214
cdb_put_data() 215
cdb_put_hdr() 216
cdb_put_image() 217
cdb_put_record() 218
cdb_put_record_comment() 219
cdb_put_record_data() 220
cdb_put_record_hdr() 221
cdb_put_response() 222
cdb_put_result() 223
cdb_record 225
cdb_seek() 228
cdb_seek_record() 229
cdb_select_acqimg() 231
cdb_select_acquire() 233
cdb_select_command() 235
cdb_select_data() 237
cdb_select_image() 239
cdb_select_record() 241
cdb_select_response() 243
cdb_select_result() 245
cff_find() 247
cff_results 248
cfifo_buffer 249
cfifo_data() 251
cfifo_dump() 252
cfifo_empty() 253
cfifo_free() 254

cfifo_get() 255
cfifo_make() 256
cfifo_put() 257
cfifo_read() 258
cfifo_write() 259
cfocus_cleanup() 260
cfocus_diff_sum() 261
cfocus_gradient() 262
cfocus_search() 263
cfs_clear_boot() 265
cfs_clear_boot_floppy() 266
cfs_clear_boot_hard() 267
cfs_clear_floppy() 268
cfs_diskabort() 269
cfs_diskclose() 270
cfs_diskopen() 271
cfs_diskopen_count() 272
cfs_format() 273
cfs_format_floppy() 274
cfs_format_hard() 276
cfs_free_space() 277
cfs_get_case() 278
cfs_get_cwd() 279
cfs_get_drive() 280
cfs_get_separator() 281
cfs_gdone() 282
cfs_gfirst() 283
cfs_gnext() 284
cfs_init() 285
cfs_mkdir() 286
cfs_partition() 287
cfs_partition_all() 288
cfs_poll_disks() 289
cfs_rmdir() 290
cfs_set_case() 291
cfs_set_cwd() 292
cfs_set_drive() 293
cfs_set_separator() 294
cfs_stat 295
cfs_util_boot_to_file() 296
cfs_util_clear_boot() 297
cfs_util_clear_boot_floppy() 298
cfs_util_clear_boot_hard() 299
cfs_util_clear_floppy() 300
cfs_util_copy() 301
cfs_util_delete() 302
cfs_util_file_to_boot() 303
cfs_util_format() 304
cfs_util_format_floppy() 305
cfs_util_format_hard() 307
cfs_util_list() 308

cfs_util_mkdir() 309
cfs_util_partition() 310
cfs_util_partition_all() 311
cfs_util_print_partition() 312
cfs_util_rename() 313
cfs_util_rmdir() 314
cfs_util_vol_to_boot() 315
cgcl_calibration 316
cgcl_check_align() 321
cgcl_check_tilt() 322
cgcl_create_calibration() 323
cgcl_delete_calibration() 324
cgcl_error_table 325
cgcl_fov_size() 326
cgcl_get_fov() 327
cgcl_image_to_op() 328
cgcl_move() 329
cgcl_op_to_image() 330
cgcl_op_to_physical() 331
cgcl_params 332
cgcl_physical_to_op() 335
cgcl_receive() 336
cgcl_set_fov() 337
cgcl_sync() 338
cgcl_transmit() 339
cght_delete_model() 341
cght_display_model() 342
cght_model 343
cght_params 345
cght_receive_model() 347
cght_results 348
cght_run() 349
cght_train() 350
cght_train_params 351
cght_transmit_model() 353
cgm_close() 354
cgm_create_element() 355
cgm_create_element_view() 356
cgm_delete_element() 357
cgm_delete_element_view() 358
cgm_dilate() 359
cgm_element 360
cgm_element_view 362
cgm_erode() 363
cgm_lib_elements 364
cgm_make_lib() 365
cgm_open() 366
cgr_caliper 367
cgr_caliper_init() 368
cgr_caliper_params 370
cgr_character() 371

cgr_circle() 372
cgr_closed 373
cgr_closed_init() 374
cgr_corner 376
cgr_corner_init() 377
cgr_cross() 379
cgr_display_image() 380
cgr_display_model() 382
cgr_ellipse() 384
cgr_fgscreen 386
CGR_FIG_IMG() 387
cgr_figure 388
cgr_fill() 390
cgr_grid() 391
cgr_gwindow 392
cgr_gwindow_init() 393
cgr_gwindow_params 395
cgr_histogram() 396
cgr_iline() 397
cgr_image() 398
cgr_line() 399
cgr_mod_dontcare 400
cgr_outline() 401
cgr_point() 402
cgr_pointer() 403
cgr_pointer_style 404
cgr_printf() 405
cgr_ruler_init() 408
cgr_screen 409
cgr_stack 412
cgr_stack_clear() 413
cgr_stack_down() 414
cgr_stack_frame 415
cgr_stack_header 416
cgr_stack_init() 417
cgr_stack_pop() 418
cgr_stack_splice() 419
cgr_stack_to_mark() 420
cgr_stack_up() 421
cgr_string() 422
cgr_string_size() 424
cgr_synthetic_closed_init() 425
cgr_synthetic_corner_init() 426
cgr_synthetic_params 427
cgr_tone() 429
cgr_video_start 431
cgr_video_stop 432
cgr_vidfont 433
cgrz_circle() 434
cgrz_coords() 435
cgrz_copy() 436

cgrz_ellipse() 437
cgrz_fill() 438
cgrz_get_scroll() 439
cgrz_get_zoom() 440
cgrz_grid() 441
cgrz_line() 442
cgrz_overrides 443
cgrz_overview() 444
cgrz_overview_mode 445
cgrz_pelvalues() 446
cgrz_point() 447
cgrz_pointer() 448
cgrz_scroll_relative() 449
cgrz_scroll_zoom() 450
cgrz_scroll_zoom_copy() 451
cgrz_set_scroll() 452
cgrz_set_zoom() 453
cgrz_setup() 454
cgrz_string() 456
cgrz_tablet 457
cgrz_zoom_in() 460
cgrz_zoom_out() 461
cgtc_acq 462
cgtc_blob() 463
cgtc_copy_module() 465
cgtc_delete_global() 466
cgtc_delete_module() 467
cgtc_delete_mresults() 468
cgtc_delete_results() 469
cgtc_dflt_map 470
cgtc_dimg 471
cgtc_eimg 472
cgtc_fimg 473
cgtc_geom 474
cgtc_init_global() 475
cgtc_load_vb_action() 476
cgtc_make_gwindow() 477
cgtc_make_timg() 478
cgtc_map_sobel() 479
cgtc_mask_xfer() 480
cgtc_meta_run() 481
cgtc_mod_params 483
cgtc_module 484
cgtc_morph() 488
cgtc_mpar 490
cgtc_mresults 492
cgtc_mrun_params 494
cgtc_new_module() 495
cgtc_receive_module() 496
cgtc_res_lst 497
cgtc_reset_vb_action() 498

cgtc_results ... 499
cgtc_rt_trn_params ... 502
cgtc_run() ... 504
cgtc_run_cst() ... 506
cgtc_run_meth ... 508
cgtc_run_params ... 509
cgtc_set_morph() ... 513
cgtc_set_st_thresh() ... 514
cgtc_set_ti_thresh() ... 515
cgtc_setup_graphics_luts() ... 516
cgtc_sobel() ... 517
cgtc_swap() ... 518
cgtc_swap_params ... 519
cgtc_tmpl_data ... 520
cgtc_train() ... 521
cgtc_transmit_module() ... 522
cgtc_trn_params ... 524
cgtc_window_module() ... 525
chlt_data ... 526
chlt_draw() ... 527
chlt_free() ... 528
chlt_input ... 529
chlt_make() ... 530
chlt_params ... 531
chlt_print() ... 534
chlt_results ... 535
chlt_run() ... 537
chp_alloc_total ... 538
chp_alloca() ... 539
chp_check ... 540
chp_free_total ... 541
chp_free_trace() ... 542
chp_maxnet ... 543
chp_print_mode ... 544
chp_print_stats() ... 545
chp_print_trace() ... 546
chp_results ... 547
chp_script ... 549
chp_statistics() ... 550
chp_stats() ... 551
chp_trace() ... 552
chp_trace_entry ... 554
chp_trace_hdr ... 555
cia_15_8 ... 556
cia_31_16 ... 557
cia_atan() ... 558
cia_coeff_matrix ... 559
cia_coord_xform() ... 560
cia_cos() ... 561
cia_cos16() ... 562
CIA_DOUBLE_TO_8() ... 563

CIA_DOUBLE_TO_16() 564
cia_float 565
CIA_FLOAT8() 566
CIA_FLOAT14() 567
CIA_FLOAT16() 568
cia_limit() 569
cia_lin_limit() 570
cia_point 571
cia_rect 572
CIA_RND8() 573
CIA_RND16() 574
cia_sin() 575
cia_sin16() 576
cia_sqrt() 577
cif_create_pd() 578
cif_cursor_stop() 579
cif_cursor_update() 580
cif_delete_pd() 581
cif_fg_cursor 582
cif_fg_gainoff 583
cif_fg_split 584
cif_fg_window 585
cif_gainoff_start() 586
cif_gainoff_stop() 587
cif_gainoff_update() 588
cif_iv_cursor 589
cif_iv_tb3 590
cif_iv_window 591
cif_magnify() 592
cif_make_cursor() 593
cif_make_gainoff() 594
cif_make_split() 595
cif_make_window() 596
cif_open_tb() 597
cif_pd 598
cif_pd_check() 601
cif_pd_start() 602
cif_pd_track() 603
cif_realtime() 604
cif_set_arms() 605
cif_set_color() 606
cif_set_cursor() 607
cif_set_window() 608
cif_split_start() 609
cif_split_stop() 610
cif_split_update() 611
cif_std_lbut() 612
cif_std_mbut() 613
cif_std_rbut() 614
cif_tb_file 615
cif_tb3_flush() 616

cif_tb3_read() 617
cif_tb3_readnw() 618
cif_track_figure() 619
cif_window_check() 620
cif_window_start() 621
cif_window_stop() 622
cif_window_track() 623
cif_window_update() 624
cio_dev_num 625
cio_echo() 626
cio_expansion_int() 627
cio_getdev() 628
cio_getdev_qset() 630
cio_gtty() 631
cio_init() 632
cio_par_params 633
cio_par_receive() 634
cio_par_transmit() 635
cio_pulse_out() 636
cio_query() 638
cio_read_dipsw() 639
cio_read_dipsw_vp() 640
cio_read_in() 641
cio_read_out() 642
cio_ser_params 643
cio_ser_receive() 644
cio_ser_transmit() 645
CIO_SET_OUT() 646
cio_stream_params 647
cio_stream_receive() 648
cio_stream_transmit() 649
cio_stty() 650
CIO_TEST_IN() 651
CIO_TEST_OUT() 652
CIO_TEST_SW() 653
CIO_TEST_SWVP() 654
cio_vme_master_space 655
cio_vme_slave_ctrl 656
cio_vme_slave_set() 657
cio_write_out() 658
cio_xfer_ascii_params 659
cio_xfer_ending 660
cio_xfer_params 661
cio_xfer_prep 662
cio_xfer_state 664
cio_xfer_style 665
cio_xread() 666
cio_xwrite() 667
cip_add() 668
cip_arith() 669
cip_bandrestore() 671

cip_binary_copy() 673
cip_buffer 675
cip_close() 678
cip_contrast() 680
cip_copy() 681
cip_cprint() 683
cip_cread() 685
cip_create() 686
cip_delete() 688
cip_dilate() 689
cip_dist_match() 691
cip_edge_data 693
cip_edge_free() 694
cip_edge_map 695
cip_edge_params 696
cip_edge_results 701
cip_edge_type 702
cip_erode() 703
cip_expand() 705
cip_fast_add() 707
cip_gauss() 708
cip_hist_stats() 710
CIP_IS_FAST8() 711
cip_laplace() 712
cip_mask_copy() 713
cip_max() 714
cip_min() 715
cip_norm_func 716
cip_nprint() 717
cip_nread() 718
cip_nsub() 719
cip_nsub_codes 721
cip_open() 722
cip_overlap() 724
cip_overlap_fg() 725
cip_paint() 726
cip_paint_close() 729
cip_paint_dark() 730
cip_paint_dilate() 731
cip_paint_erode() 732
cip_paint_filter() 734
cip_paint_light() 735
cip_paint_mean() 736
cip_paint_median() 737
cip_paint_open() 738
cip_peak() 739
cip_peak_area 741
cip_peak_entry 742
cip_peak_free() 743
cip_peak_list 744
cip_peak_params 745

cip_peak_type 747
cip_pixel_map() 748
cip_polar_create() 749
cip_polar_delete() 751
cip_polar_map 752
cip_polar_receive() 753
cip_polar_results 754
cip_polar_size 755
cip_polar_transform() 756
cip_polar_transmit() 757
cip_quick() 759
cip_quick_init() 761
cip_rank() 762
cip_receive_image() 765
cip_rotate() 766
cip_rotate_centered() 767
cip_row_update() 768
cip_sample() 769
cip_scale() 771
cip_separable_median() 772
cip_set() 773
cip_sobel() 774
cip_sobel_init() 776
cip_stats() 777
cip_stats_params 778
cip_stats_results 779
cip_sub() 780
cip_tails() 781
cip_tails_copy() 782
cip_transform() 783
cip_transform_cleanup() 785
cip_transmit_image() 786
cip_transpose() 788
cip_vb_copy() 789
cip_window() 790
cip_zhist() 792
clf_acq_rate() 793
clf_acq_rate_results 794
clf_delete_buffer() 795
clf_delete_roi() 796
clf_get_buffer() 797
clf_interrupt_acq() 798
clf_make_arb_region() 799
clf_make_region() 801
clf_region_params 802
clf_roi 803
clf_set_buffer() 804
CLR_NOTOVFL() 805
CLS_CARD_ADDR() 806
cls_card_params 807
cls_context 808

cls_init_card() ... 812
cls_init_pos() ... 813
cls_internal_enc() ... 814
cls_lut_fill() ... 815
cls_lut_inv() ... 816
cls_lut_pos() ... 817
cls_lut_read() ... 818
cls_lut_set() ... 819
cls_set_lut_video() ... 820
cmap_function ... 821
cmap_make() ... 822
cmap_map_data ... 823
cmap_params ... 826
cmap_results ... 827
cmap_tails() ... 829
cmenu_append_data() ... 830
cmenu_append_item() ... 831
cmenu_bar ... 832
cmenu_data ... 834
cmenu_init_bar() ... 840
cmenu_insert_menu() ... 841
cmenu_item ... 842
cmenu_menu ... 844
cmenu_redisplay() ... 846
cmenu_remove_item() ... 847
cmenu_remove_menu() ... 848
cmenu_run() ... 849
cmenu_set_item() ... 850
cmenu_set_menu() ... 851
cmorph_3x3() ... 852
cmorph_3x3_filter() ... 853
cmorph_close() ... 854
cmorph_dilate() ... 856
cmorph_element ... 857
cmorph_element_type ... 859
cmorph_erode() ... 860
cmorph_op ... 862
cmorph_open() ... 863
cmorph_params ... 865
cnvm_alloc() ... 866
cnvm_compact() ... 867
cnvm_consistent() ... 868
cnvm_free() ... 869
cnvm_get_id() ... 870
cnvm_get_ptr() ... 871
cnvm_get_size() ... 872
cnvm_id_used() ... 873
cnvm_init() ... 874
cnvm_init_ptr() ... 875
cnvm_next_alloced() ... 876
cnvm_statistics() ... 877

cof_clear() 878
cof_push() 879
cof_screen 880
cof_write() 881
copto_init() 882
copto_pulse_out() 883
copto_read_in() 884
copto_read_mode() 885
copto_read_out() 886
copto_write_mode() 887
copto_write_out() 888
CPRO_BINS() 889
cpro_project() 890
cpro_weights() 891
cpsi_add() 892
cpsi_fwd() 893
cpsi_geom 894
cpsi_init() 895
cpsi_polar() 896
cpsi_rev() 897
cpsi_solve() 898
crdr_as_params 899
crdr_as_results 900
crdr_cal_data 901
crdr_char_results 902
crdr_chk_cognex() 903
crdr_chk_semi() 904
crdr_cleanup() 905
crdr_comp_as() 906
crdr_compute_calibration() 907
crdr_constraints 909
crdr_control 910
crdr_delete_font() 911
crdr_display_fld() 912
crdr_edge_data 913
crdr_ff_config 914
crdr_ff_pos 915
crdr_ff_results 916
crdr_field_finder() 917
crdr_fielding_tb() 918
crdr_font 919
crdr_generate_string() 920
crdr_init() 921
crdr_ink_type 922
crdr_model 923
crdr_op 924
crdr_params 925
crdr_point_read() 929
crdr_polarity_map[] 930
crdr_process_image() 931
crdr_set_fld() 932

crdr_string() 933
crdr_string_results 935
crdr_train_font() 937
crdr_train_params 938
crip_block 939
crip_buffer 940
crip_connectivity() 941
crip_delete() 942
crip_image_copy() 943
crip_image_set() 944
crip_params 945
crip_region 946
crip_region_zhist() 947
crip_run 948
crip_window_copy() 949
crsw_area_search() 950
crsw_init() 952
crsw_params 953
crsw_search() 955
crsw_stats 957
crsw_update() 958
crv_chk_cognex() 959
crv_chk_semi() 960
crv_constraints 961
crv_delete_box_results() 962
crv_delete_font() 963
crv_delete_font_all() 964
crv_delete_model() 965
crv_dump_font() 966
crv_fdump_font() 967
crv_fload_font() 968
crv_font 969
crv_load_font() 970
crv_make_constraints() 971
crv_make_font() 973
crv_op 974
crv_params 975
crv_point_read() 979
crv_point_recognize() 980
crv_point_verify() 982
crv_read() 983
crv_receive_font() 984
crv_recognize() 985
crv_results 986
crv_string_recognize() 987
crv_string_results 989
crv_transmit_font() 990
crv_verify() 992
cse_area_search() 993
cse_contrast() 994
cse_line_search() 995

cse_model ... 996
cse_model_type ... 1000
cse_params ... 1001
cse_pm_control ... 1003
cse_point_search() ... 1004
cse_polar_results ... 1005
cse_polar_search() ... 1006
cse_results ... 1007
cse_search_control ... 1009
csi_count() ... 1010
csi_define() ... 1011
csi_disable() ... 1012
csi_enable() ... 1013
csi_send() ... 1014
csn_angle() ... 1015
csn_angle_params ... 1016
csn_angle_results ... 1017
css_clean_memory() ... 1018
css_compress_font() ... 1019
css_eof ... 1020
css_save_system() ... 1021
css_status() ... 1026
css_uncompress_font() ... 1028
csys_backplane ... 1029
csys_board_type ... 1030
csys_boot_ramsize() ... 1031
csys_bp_type ... 1033
csys_clock_config ... 1034
csys_config ... 1035
csys_eprom_type ... 1038
csys_fpproc_type ... 1039
csys_get_config() ... 1040
csys_identify_backplane() ... 1041
csys_opt1 ... 1042
csys_proc_type ... 1043
csys_read_status() ... 1044
csys_rev_type ... 1045
csys_set_boot_type() ... 1046
csys_set_master_cfg() ... 1047
ctg_clear() ... 1048
ctg_count() ... 1049
ctg_init() ... 1050
ctg_inuse() ... 1052
ctg_release() ... 1053
ctg_state() ... 1054
ctm_begin() ... 1055
ctm_delay() ... 1056
ctm_read() ... 1057
ctm_timer ... 1058
ctr_candidate ... 1059
ctr_contrast() ... 1060

ctr_corner() 1061
ctr_delete_line() 1063
ctr_delete_model() 1064
ctr_dump_model() 1065
ctr_edge() 1066
ctr_fdump_model() 1067
ctr_fload_model() 1068
ctr_geom_score() 1069
ctr_is_binary() 1070
ctr_line() 1071
ctr_load_model() 1072
ctr_min_timeout() 1073
ctr_origin() 1074
ctr_params 1075
ctr_polar_model() 1079
ctr_polar_results 1081
ctr_receive_model() 1082
ctr_results 1083
ctr_select_model() 1084
ctr_select_model_params 1085
ctr_space 1087
ctr_train_model() 1088
ctr_transmit_model() 1091
ctr_unique_score() 1093
ctr_weighted_score() 1094
cu_bin_to_deg() 1095
cu_clear() 1096
cu_copy() 1097
cu_deg_to_bin() 1098
cu_expand_stack() 1099
cu_load_tick() 1100
cu_load_vb() 1101
cu_load_watch() 1102
cu_reset_watch() 1104
cu_version 1105
CURSOR 1106
CURSOR_X and CURSOR_Y 1107
cvc_alloc() 1108
cvc_codebufsize 1109
cvc_compile() 1110
cvc_debug_fn() 1111
cvc_init() 1112
cvc_max_file_errors 1113
cvc_max_file_warns 1114
free() 1115
GAINOFF 1116
IMAGE 1117
M 1118
malloc() 1119
max() 1120
MAX() 1121

min() 1122
MIN() 1123
par_fsetup() 1124
par_read() 1126
par_setup() 1127
par_shutdown() 1129
par_write() 1130
realloc() 1131
RT 1132
SET_NOTOVFL() 1133
setbuf() 1134
setbuffer() 1135
setlinebuf() 1136
sgttyb 1137
SPLIT 1140
VRT 1141

MACRO: **BOX** and **I**

PURPOSE: Use the trackball to change the size or position of a window of the frame grabber; i.e., *cif_fg_window->function_iv->imbuf* window.

REFERENCE: System Software: Interface

GLOBALS: *caq_image*
cif_fg_window

USE:
```
#define BOX cif_fg_window->track(cif_fg_window)
#define I BOX
```

NOTES: BOX runs the standard Cognex window trackball. *cif_fg_window->track* points to the function **cif_window_track()**. The *cif_fg_window->function_iv* instance variable points to a **cif_iv_window** structure. The structure contains three entries: one for window color (black, white, or invisible), one for the trackball mode (position or size), and one which points to a **cip_buffer** which is a window of the frame grabber.

Clicking the middle trackball button switches between position and size modes. Clicking the left button exits. Clicking the right button has no effect.

BOX and **I** are identical; **I** is provided as a shortened notation for **BOX**.

TYPE: **c_function**

REFERENCE: System Software: V Compiler

INCLUDE FILE: *standard.h*

DEFINITION:
```
typedef int (*c_function) ();
```

NOTES: This type is used in Cognex-defined data structures.

FUNCTION: **caq_acquire()**

PURPOSE: Acquire an image.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

GLOBALS: *caq_image, caq_even, caq_odd, caq_stored_image_polarity.*

USE:

```
cip_buffer *caq_acquire(mode)
   caq_context *mode;
```

NOTES: **caq_acquire()** acquires and returns a pointer to an image: the full frame, the even field, or the odd field. **caq_acquire()** puts the frame grabber in acquire mode and then leaves the frame grabber in CPU mode.

mode: points to an instance of a **caq_context** structure, which contains parameters to control the acquisition. If the *x_res*, *y_res*, and *image_buffer* fields are the same for both the default display (*caq_mode*) and the **caq_context** pointed to by *mode*, then **caq_acquire()** will return one of the three pointers: *caq_image*, *caq_even*, or *caq_odd*. If any of these fields specified by *caq_mode* is different than that specified in the *caq_context* pointed to by *caq_mode*, then the pointer returned is not necessarily equal to one of *caq_image*, *caq_even*, or *caq_odd* and the resolution or image buffer is changed just for the duration of the acquire.

The acquisition is completed and the function returns within 51 ms when using RS-170 video timing, and within 60 ms when using CCIR video timing.

If you select software-assisted acquisition (by setting the global variable *caq_format* to *caq_format_general_softacq*), **caq_acquire()** copies the image to the specified destination image buffer (allocated in main memory) and returns a pointer to the image *in the destination buffer*. You must allocate the destination buffer beforehand using **cip_create()** and identify it using **clf_set_buffer()**. Also, **caq_acquire()** does not necessarily set **caq_image** to point to the acquired image.

You must allocate memory space greater than or equal to the size of the region of interest; any part of the region of interest that falls outside of the allocated memory space is lost. **caq_acquire()** does not throw an error if this is the case.

If you anticipate that the image will fit entirely within the hardware frame buffer, you may disable software-assisted acquisition by setting the global variable *caq_format* to *caq_format_general*.

caq_acquire() throws the following errors:

- *CAQ_ERR_MOVING* if the system is *armed* or *triggered* and trigger 2 is not specified, or if trigger 2 is specified and the moving acquisition state is *off* or *processing*

- *CAQ_ERR_NOTINUSE* if a trigger in one of the following ranges is specified (acquire after move) but the trigger has not been set up with a call to **ctg_init()**:
  - 3 - 6 on a 2000/3000
  - 3 - 8 on a 4400
  - 3 - 10 on a 4100/4200 EX
  - 3- 4 on a 5000
- *CAQ_ERR_FORMAT* if the fields *x_res*, *y_res*, *image_buffer*, and *graphics_buffer* for the acquisition pointed to by *mode* are not a valid combination
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode
- *CGEN_ERR* if you have specified a software-assisted acquisition and you have not specified a region of interest.
- *CGEN_ERR* if you have not allocated a destination image buffer in 4400 main memory.

FUNCTION: **caq_arm()**

PURPOSE: Enable a moving acquisition for use with **caq_acquire()**.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:
```
int caq_arm()
```

NOTES: **caq_arm()** enables a moving acquire and returns the number of overruns; that is, the number of triggers not resulting in acquires because the system was in the *triggered* or *processing* state.

On the 2000, **caq_arm()** sets the protection for frame memory to "no access."

caq_arm() throws the following errors:

- *CAQ_ERROR_MOVING* if the state is not *triggered* or *processing*
- *CAQ_ERROR_ASYNC* if the system is in asynchronous mode

FUNCTION: **caq_async_end()**

PURPOSE: Changes the status of a linked list to inactive. If this is the last active list, the system returns to synchronous mode.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:
```
caq_async_end(context_p)
   caq_context *context_p;
```

NOTES: **caq_async_end()** changes the status of a list of **caq_context** structures to inactive. If this is the last active list, the system returns to synchronous mode.

context_p: points to a list of active **caq_context** structures.

caq_async_end() throws the following error:

- *CGEN_ERR_BADARG* if *context_p* does not point to an active list

FUNCTION: **caq_async_init()**

PURPOSE: Activates a linked list of **caq_context** structures for asynchronous acquisition. Put the system in asynchronous mode.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:
```
caq_async_init(context_p)
   caq_context *context_p;
```

NOTES: **caq_async_init()** allows you to activate a list of **caq_context** structures in order to allow asynchronous changes to trigger and acquisition specifications. Once the system has a list available it is in asynchronous mode until there are no more active lists.

context_p: a pointer to a linked list of one or more **caq_context** structures.

Each **caq_context** structure has a *next* component that points to the next **caq_context** structure in the linked list, or that is NULL. Once there are no more structures in this list, the list becomes inactive. A circular list (a linked list whose last structure points to its first structure) will run until you call **caq_async_end()**. Because of hardware limitations on the 2000, *caq_context->next* must always be NULL.

On the 2000, you must change frame grabber modes to control access to the frame store. You should not call **caq_async_init()** while the frame grabber is in CPU mode; call **caq_display()**, **caq_graphics()**, or **caq_video()** to change the frame grabber mode. Call **caq_cpu()** to return to CPU mode after the acquisition is complete.

Applications can define a maximum of four lists.

caq_async_init() throws the following errors:

- *CVC_ERR_ALLOC* if insufficient heap space is available during validation of the **caq_context** structures
- *CAQ_ERR_FULL* if you attempt to define more than four lists
- *CAQ_ERR_INUSE* if the list pointed to by *context_p* is already in use
- *CGEN_ERR_BADARG* if an error is detected in any **caq_context** structure
- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*

VARIABLE: **caq_blink_enable**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
bool caq_blink_enable;
```

NOTES: *caq_blink_enable* (available only on the 3000/4000/5000) specifies whether pixels of the graphic value *CAQ_BLINK* are displayed simply as the color assigned to *CAQ_BLINK*, or whether they blink between the color assigned to *CAQ_BLINK* and the color assigned to *CAQ_BLACK*.

- Setting *caq_blink_enable* to zero, causes all *CAQ_BLINK* pixels to be displayed, *without blinking*, in the color assigned to *CAQ_BLINK*.
- Setting *caq_blink_enable* to one, causes all *CAQ_BLINK* pixels to blink between the value of *CAQ_BLINK* and the value of *CAQ_BLACK*.

The following information applies only to the 5000. On the 5000, setting *caq_blink_enable* to 1 causes the graphics pixels displayed on the RGB monitor connected to the 5000 to blink. This will not cause graphics pixels to blink on the VGA monitor connected to the video mixing board. By default, all graphics pixels assigned to the graphics value *CAQ_BLINK* are displayed as cyan by the video mixing board, regardless of the value of *caq_blink_enable*.

FUNCTION: **caq_cancel()**

PURPOSE: Cancels a moving trigger setup (for moving acquisition using **caq_acquire()**) and sets the moving trigger state to off.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:
```
void caq_cancel()
```

NOTES: The state may be *armed*, *triggered*, or *processing*.

On the 2000, **caq_cancel()** sets the protection for frame memory to "no access."

Trigger event 2 will no longer cause an acquisition when the moving trigger state is off. The trigger event will still be recognized and will cause the associated action routine, if any, to run until it is released by calling **ctg_release(***2***)**.

caq_cancel() throws the following errors:

- *CAQ_ERROR_ASYNC* if the system is in asynchronous mode
- *CAQ_ERR_MOVING* is if the state is *off*

FUNCTION: **caq_constant()**

PURPOSE: Load the entire frame store with a constant value.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_constant(z)
   int z;
```

NOTES: *z:* the grey value with which to fill the frame store.

caq_constant() loads the entire frame store with a constant grey value. On the 2000, that value may be in the range 0 - 63. On the 3000/4400, that value may be in the range 0 to 255. **caq_constant()** leaves the frame grabber in CPU mode.

caq_constant() may be called at any time for synchronous stationary acquisition and acquisition after move, while in the processing state for synchronous moving acquisition, and may not be called while asynchronous acquisition is active.

caq_constant() affects only the image currently designated by *caq_mod->image_buffer*; it does not affect any other image buffers. The function does not affect the graphics plane.

caq_constant() throws the following errors:

- *CIP_ERR_PELDEPTH* if *z* exceeds 6 bits on the 2000, or 8 bits on the 3000/4000/5000.
- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode

STRUCTURE: **caq_context**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
typedef struct caq_context {
   unsigned char
      gain,offset,
      trigger,settle,strobe,camera,
      acquire_polarity,display_polarity;
   short display_value;
   caq_frame field;
   short split,cti_start[2],cti_end[2];
   char
      x_res,y_res,
      image_buffer,graphics_buffer,
      eightbit_enable,gplane_enable,
      input_control,display_control;
   unsigned char state;
   struct caq_context *next;
   short active_time1,active_time2;
   int *acquire_overrun_cnt,*trigger_overrun_cnt;
   char *camera_specific;
   clf_roi *region;
} caq_context;
```

NOTES: A **caq_context** structure defines the parameters of the frame grabber for acquisition and display.

gain: determines the dynamic range of the digital values that will be assigned to the video signal in the A/D converter. Adjust the value of *gain* when you want to change the contrast of an acquisition. *gain* can have values in the range 0-255, with lower values representing higher gain in the analog section. A typical value is *CAQ_STD_GAIN*.

If the gain is less than optimum, the digital values will have a smaller distribution and small variations in signal amplitude will be assigned the same digital value. This may cause important low-contrast information to be lost. If the gain is greater than optimum, one or both ends of the range may be clipped.

The values you select for *gain* and *offset* are not portable among the 2000, 3000, 4100/4200, 4400, and 5000 vision processors. If you want to write code that will run on all machines, you should use the constants *CAQ_STD_GAIN* and *CAQ_STD_OFFSET* as values for *gain* and *offset*. These constants have appropriate values on each machine.

You should not use the constants *CAQ_STD_GAIN* and *CAQ_STD_OFFSET* for line-scan acquisitions. Their settings for the LSO-1000 are as follows:

For line-scan acquisitions, *gain* is an integer from 0 to 255, specified according to the following equation:

$$V = 0.0156 \times A \times G \times I$$

$$G = \frac{V}{0.0156 \times A \times I}$$

where:

G = gain

V = resulting voltage of the A/D converter

A = attenuation factor, set via jumpers on the LSO-1000 module:
+1.0 with a 3.0dB attenuation pad and non-inverting amplifier
-1.0 with a 3.0dB attenuation pad and inverting amplifier
+0.4 with a 7.0dB attenuation pad and non-inverting amplifier
-0.4 with a 7.0dB attenuation pad and inverting amplifier

I = voltage of the video input channel

The range of available gain when using the 3.0 dB pad is 0.0 to 4.0.

The range of available gain when using the 7.0 dB pad is 0.0 to 1.6.

The range of available gain when using differential input is 0.0 to 4.0.

offset: is the lower limit of the digital values that will be assigned to the video signal in the A/D converter. Change *offset* when you want to change the brightness of an acquisition. A typical value is *CAQ_STD_OFFSET.*

Adjusting the offset lets you select the lower and upper limits saturation points) at which differentiation of assigned digital values begins and ends. Due to variations in camera components, the lowest digital value that can be converted may not be precisely 0, but will be close to 0. All signal voltages that exceed the upper limit of the dynamic range are assigned the highest digital value possible in the specified number of bits per pixel.

offset is not portable among the 2000, 3000, 4100/4200, 4400, and 5000 vision processors. If you want to write code that will run on all machines, you should use the constant *CAQ_STD_OFFSET* as a value for *offset*. This constant has appropriate values on each machine.

For line-scan acquisitions, *offset* is an integer from 0 to 255, the resulting voltage of the A/D converter is:

$$V = 5.2 \times \left(\frac{L}{128} - 1 \right)$$

where:

V = resulting voltage of the A/D converter

L = offset level

The possible range of DC offset available is -5.2 to +5.2 VDC.

trigger: the input line that triggers acquisition on the vision processor.

For synchronous acquisitions using **caq_acquire()** you can specify the following values:

- 0, for stationary acquisition
- 2, for moving acquisition
- 3 - 6, for acquisition after move operations. (The 4400 can have values 3 - 8.)

You may not use line 1 to trigger acquisitions using **caq_acquire()**.

For asynchronous acquisition using **caq_async_init()**, *trigger* can be 0 - 6 for a Cognex 2000/3000, 0 or 1 for a 4100/4200, 0 - 8 for a 4400 or a 4100/4200 EX, and 0 - 3 for a 5000.

On the 4000/5000, in asynchronous mode only, trigger may also have the value *CTG_TRIGGER_BUS*, which specifies a trigger generated by a host on the VMEbus.

settle: the total time, in 60^{ths} of a second (RS-170) or 50^{ths} of a second (CCIR) to wait after a move is complete before an acquisition may begin. *settle* is used only in synchronous acquisition after move operations (in which case *trigger* should be 3 - 6 for a 2000/3000, or 3 - 8 on a 4400), and in asynchronous acquisition when *trigger* is nonzero.

The standard value of *settle* is 2, which is the integration time of the sensor in a standard solid state camera. You may need to set it higher if the scene to be acquired vibrates for a period of time after the move is complete. For asynchronous acquisition, the value for *settle* must be less than or equal to the value of *active_time1*.

strobe: the TTL output line to use to fire a strobe. You can specify values 0 - 12 on a 2000/3000, 0 - 1 on a 4100/4200, 0 - 8 on a 4400 or a 4100/4200 EX, and 0 - 3 on a 5000. The strobe is triggered by the corresponding output line. Specify 0 if you are not using a strobe.

A strobe is typically required to freeze an image for a moving acquisition, but you may use a strobe for any type of acquisition.

camera: specifies the camera(s) to use for image acquisition. It may have values 1-4.

On the 2000/3000, *camera* operates in conjunction with *split*, described below (*split* is ignored on the 4000/5000). If *split* is set to 0, the value specified for *camera* selects the corresponding camera. If *split* is set to a nonzero value:

- setting *camera* to 1 specifies a camera 1/camera 2 split;
- setting *camera* to 3, specifies a camera 3/camera 4 split.

When you perform any acquisition using a line-scan module, *camera* must be set to the integer index for the line-scan module specified in your call to **cls_init_card()**.

acquire_polarity: the polarity of acquired images: *positive* or *negative*.

A nonzero value specifies positive images: black is mapped to pixel value 0, and white is mapped to pixel value 2^n-1, where *n* is the number of bits per pixel.

A value of 0 specifies negative images. The negative value, *V*, of a pixel value *P*, having *n* bits per pixel, is defined as follows:

$$V = |P-(2^n-1)|$$

Intuitively, this is the opposite of the pixel's grey-scale value.

The value of the global variable *caq_stored_image_polarity* is set to *acquire_polarity* when an acquisition takes place.

display_polarity: the polarity of the frame grabber's display: *positive* or *negative*. The display polarity of an image has no effect on the stored pixel values.

A nonzero value specifies positive images: black is mapped to pixel value 0, and white is mapped to pixel value 2^n-1, where *n* is the number of bits per pixel.

A value of 0 specifies negative images. The negative of a pixel value is defined in the same way as for the *acquire_polarity* field, above.

If you are using the 5000 with the video mixing board, you should refer to the *Acquisition and Display* chapter of the *Image Processing* manual for more details on properly setting *eightbit_enable* for the 5000. Also refer to the *ISA Bus I/O* chapter of the *System Software* manual for a description of the 5000's video lookup table.

display_value: specifies what is displayed on the monitor when the frame grabber is in CPU mode: either a constant grey value or raw video. *display_value* may have values -1 through 63 (-1 through 255 on the 3000/4000/5000).

Specifying -1 displays raw video; specifying a value from 0 (black) to 63/255 (white) displays that grey value.

If you are using the 5000 with the video mixing board, you should refer to the *Acquisition and Display* chapter of the *Image Processing* manual for more details on properly setting *eightbit_enable* for the 5000. Also refer to the *ISA Bus I/O* chapter of the *System Software* manual for a description of the 5000's video lookup table.

field: specifies the image fields to be acquired by the frame grabber. A field is a scan of every other line over a camera's entire field of view. *field* must be set to a value in the following enumeration:

```
typedef enum {
   caq_full,
   caq_odd_field,
   caq_even_field,
   caq_next
} caq_frame;
```

- *caq_full* specifies acquisition of the full frame image (both even and odd fields).
- *caq_odd_field* specifies acquisition of the odd field of the image.
- *caq_even_field* specifies acquisition of the even field of the image.
- *caq_next* tells the frame grabber to acquire the next field: either even or odd.

If you want a single field and do not need to acquire either the even or odd field specifically, selecting *caq_next* may save acquisition time.

The global variable *caq_stored_field* is determined by the value of *field* when an acquisition takes place.

split: specifies whether the video image should be split between two cameras (*split* has meaning on the 2000/3000 only, and is ignored on the 4000/5000). A value of 0 signifies no split. If a split is selected (nonzero value), it specifies the line number of the video image at which the source of the image changes.

split works in conjunction with *camera*.

cti_start: the starting line numbers of the charge transfer intervals for solid state cameras. The charge transfer interval is that period of time during which the charge representing the intensity values in the scene is transferred from the individual sensor elements to an output register. Firing the strobe during the charge transfer intervals has unpredictable results.

cti_start is relevant only for acquisitions initiated by a fast trigger (trigger lines 1 and 2).

cti_start[0] is the starting line number of the even field charge transfer interval. *cti_start[1]* is the starting line number of the odd field charge transfer interval.

The acquisition software will not flash the strobe during the charge transfer interval.

cti_end: the ending line numbers of the charge transfer intervals for solid state cameras.

cti_end is relevant only for acquisitions initiated by a fast trigger (trigger lines 1 and 2).

cti_end[0] is the ending line number of the even field charge transfer interval. *cti_end[1]* is the ending line number of the odd field charge transfer interval.

Consult your camera's specification sheet for the charge transfer interval ending line number.

x_res and *y_res:* specify full or half resolution frame grabber operation. *x_res* and *y_res* may have values 0, 1 or 2. Values of 0 (for upward compatibility) and 1 specify a full resolution operation; a value of 2 specifies a half resolution operation.

On the 2000/3000, *x_res* and *y_res* may be different; that is, an acquisition can be full resolution in one dimension and half resolution in the other dimension. On the 4100/4200, 4400, and 5000, *x_res* and *y_res* are *not* independent, and specify the same resolution: an acquisition is either full resolution in both dimensions or half resolution in both dimensions.

Note that half resolution operation is not possible for large format or line-scan acquisitions (4400 with LSO-1000 module only).

Typically, you modify *caq_mode->x_res* and *caq_mode->y_res* by calling **caq_resolution()**; you typically do not modify them directly.

When set to 2, *x_res* halves the pixel sampling rate and line length. *x_res* applies to both image and graphics memory.

When *y_res* is set to 2, the even and odd video fields (*caq_even, caq_odd*) are stored in the same place in video memory. These fields apply only to image memory. They do not apply to 3000 graphics memory which is always half resolution in the y dimension.

When acquiring an image, if *y_res == 1* (full resolution), normally the *field* component will be *caq_full*. If *y_res == 2* (half resolution), normally the *field* component will be *caq_even_field*, *caq_odd_field*, or *caq_next*.

image_buffer: specifies which **cip_buffer** in the frame store to use for acquisition or display. Typically, you modify *caq_mode->image_buffer* by calling **caq_resolution()** or **caq_image_select()**.

image_buffer may have values 0 through *n* - 1, where *n* is the number of distinct buffers supported, given the image format and resolution.

graphics_buffer: specifies which 3000/4000/5000 graphics buffers to display. The 2000 ignores this component. Typically, you modify *caq_mode->graphics_buffer* by calling **caq_resolution()** or **caq_graphics_select()**; you generally do not modify it directly.

graphics_buffer may have values 0 through *n* - 1, where *n* is the number of distinct buffers supported, given the format and resolution.

eightbit_enable: controls the effective depth of the 3000/4000/5000 frame stores during acquisition (the 2000 ignores this component). *eightbit_enable* may have values 0 (disabled) or 1 (enabled).

If *eightbit_enable* is set to 1, then 8 bits are digitized and written to frame memory.

If *eightbit_enable* is set to 0, then the 6 most significant bits of the digitized video data are acquired into the 6 least significant bits of the frame memory and the 2 most significant bits are set to 0.

The value of the global variable *caq_stored_eightbit* is set to *eightbit_enable* when an acquisition takes place.

If you are using the 5000 with the video mixing board, you should refer to the *Acquisition and Display* chapter of the *Image Processing* manual for more details on properly setting *eightbit_enable* for the 5000. Also refer to the *ISA Bus I/O* chapter of the *System Software* manual for a description of the 5000's video lookup table.

gplane_enable: determines where graphics information comes from in graphics mode. *gplane_enable* may have values 0 (disabled) or 1 (enabled). If *gplane_enable* is 0, the real-time image will be overlaid with graphics from the frame store. If *gplane_enable* is 1, the real-time image will be overlaid with graphics from a graphics buffer pointed to by *graphics_buffer*.

You must use **caq_set_gplane_enable()** to set the value of *caq_mode->gplane_enable* . Do not set *caq_mode->gplane_enable* directly.

The 2000 ignores this component.

input_control and *display_control:* reserved for future use. Set these components to 0.

state: the current status of this **caq_context** structure. State may be one of the following:

CAQ_STATE_ACQUIRING: An acquisition is in progress as specified by this **caq_context** structure. This state should only be set by the acquisition software.

CAQ_STATE_INACTIVE: This **caq_context** structure is not currently in use, and is ready for an acquisition. This state is not set by acquisition software. You set it to initialize a **caq_context** structure for acquisition.

CAQ_STATE_IMAGE_AVAILABLE: An acquisition has completed as parameterized in this **caq_context** structure. This state should only be set by the acquisition software.

CAQ_STATE_LOCKED: Use of this **caq_context** structure is suspended until *state* is set to *CAQ_STATE_INACTIVE*. If *state* is *CAQ_STATE_LOCKED* for a **caq_context** structure that is at the head of its linked list, the use of the list is suspended. This state is not set by acquisition software. You can set it to lock a list.

CAQ_STATE_LOST: Acquisition failed due to timeout. This state should only be set by the acquisition software.

next: points to the next **caq_context** structure in this list, or NULL if this is the last.

active_time1: the initial value for a time period (specified as a number of vertical blanking interrupts) during which an acquisition can take place. For non-triggered acquisition, the timer starts as soon as *state* is set to *CAQ_STATE_INACTIVE*. For triggered acquisition, the timer starts as soon as *state* is set to *CAQ_STATE_INACTIVE* and a trigger occurs. If the acquisition is of the full frame image (both even and odd fields), *active_time1* provides the time out value for the first field acquired, and the field *active_time2* (below) provides the time out value for the second field acquired. *active_time1* must be greater than or equal to *settle*.

active_time2: the initial value for a time period (specified as a number of vertical blanking interrupts) during which the second field of a two field acquisition can complete. This field is meaningful only if multiple linked lists are available and the structure requires a two field acquisition.

acquire_overrun_cnt: if non-NULL, points to an **int** that contains the number of acquisitions that have been missed. Acquisitions can be missed when multiple acquisitions are active at the same time. *acquire_overrun_cnt* will correspond directly to the number of times the *state* field was set to *CAQ_STATE_LOST*.

trigger_overrun_cnt: if non-NULL, points to an **int** that contains the number of triggers that have been missed since the last trigger was armed.

camera_specific: pointer to a **cls_context** data structure containing parameters specific to the current type of camera. If not applicable, this element should be set to NULL.

region: pointer to a **clf_roi** structure which defines the region of interest for large-format acquisitions. If not applicable, this component should be set to NULL.

FUNCTION: **caq_cpu()**

PURPOSE: Put the frame grabber in CPU mode.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_cpu()
```

NOTES: On the 2000, **caq_cpu()** displays either real-time input or a constant value as selected by *caq_mode->display_value.*

The behaviors of the 3000/4000/5000 depend additionally on the value of *caq_video_mode*. If *caq_video_mode* is equal to zero, the system behaves the like a 2000. If *caq_video_mode* is equal to one, then the display does not change.

caq_cpu() throws the following errors:

- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode (2000 only)

FUNCTION: **caq_display()**

PURPOSE: Puts the frame grabber in display mode.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:
```
void caq_display()
```

NOTES: The monitor shows the contents of the frame grabber as a frozen grey level image.

caq_display() throws the following errors:

- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode (2000 only)

VARIABLE: **caq_even**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
cip_buffer *caq_even;
```

NOTES: *caq_even* points to the special **cip_buffer** corresponding to the even field of the frame store image. The pixel at coordinates (x,y) in *caq_even* is the same as that at (x,2y+1) in *caq_image.*

VARIABLE: **caq_even_line**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:
```
int caq_even_line;
```

NOTES: *caq_even_line* holds the number of the first line of *caq_even*. *caq_even_line* is initialized by **caq_init()** based on the specified image format.

VARIABLE: **caq_field**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
unsigned int caq_field;
```

NOTES: The low order bit of *caq_field* indicates whether the current video field is odd or even. *caq_field* is incremented at each vertical blanking time.

caq_field is used internally for timing and image acquisition synchronization.

VARIABLE: **caq_format**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
enum caq_format_type caq_format;
```

NOTES: *caq_format* indicates which video timing and image format to use. The variable is of type **caq_format_ type**. Images are always regular for acquisitions into the 3000 and 5000 frame grabbers. Images are irregular in the 2000 frame grabber, the 3000 graphics buffer, and in the 4400 frame grabber unless otherwise noted below.

```
typedef enum caq_format_type {
   caq_format_rs170_576,
   caq_format_rs170_512,
   caq_format_rs170_wide,
   caq_format_pal,
   caq_format_pal_512
   caq_format_general,
   caq_format_general_softacq
} caq_format_type;
```

caq_format_rs170_576 specifies 60 Hz timing and a 576 (pixels per line) x 448 (lines) image format.

caq_format_rs170_512 specifies 60 Hz timing and a 512 x 480 image format. This video format guarantees regular images in the 4400 frame grabber.

caq_format_rs170_wide specifies 60 Hz timing and a 640 x 480 format (available only on the 3000/4400).

caq_format_pal specifies 50 Hz timing and a 760 x 512 format (available only on the 3000/4400).

caq_format_pal_512 specifies 50 Hz timing and a 512 x 512 format (available only on the 3000/4400). This video format guarantees regular images in the 4400 frame grabber.

caq_format_general specifies that the video format information specified by the function **caq_specify_general_format()** be used when **caq_init()** is called (available only on the 4400 with an LSO-1000 and line-scan camera, or the 4400 with a large-format camera).

caq_format_general_softacq specifies that the video format information specified by the function **caq_specify_general_format()** be used when **caq_init()** is called. *caq_format_general_softacq* also specifies that a software-assisted acquisition will be performed with this format, and an alternate set of image buffers in main memory will be used (available only on the 4400 with LSO-1000).

Any change to *caq_format* takes effect on the next call to **caq_init()**.

Note: Whenever you change video formats, you should reset *cif_fg_window->xhi* and *cif_fg_window->yhi*, which control the limits of where the cursor can appear on the monitor. Typically, you set these to the video format's width and height.

The following information applies only to the 5000 VP: most 5000 applications set *caq_format* during initialization and do not change it. You must make sure the video mixing board is set to the corresponding mode. Refer to the *Acquisition and Display* chapter of the *Image Processing* manual for more details.

ENUMERATION: **caq_frame**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
typedef enum{
   caq_full,
   caq_odd_field,
   caq_even_field,
   caq_next
}caq_frame;
```

NOTES: Data type of the *field* component of the **caq_context** data structure and of the *caq_stored_field* variable.

In the data structure **caq_context,** *field* specifies whether the image to be acquired should be full resolution (the full frame) or half resolution (the even field only or the odd field only).

The variable *caq_stored_field* is set to a value of **caq_frame** to indicate the type of acquire that was just performed.

caq_full: the frame grabber does a full resolution acquire; that is, it acquires the full frame image (both even and odd fields).

caq_odd_field: the frame grabber does a half resolution acquire of the odd field of the image.

caq_even_field: the frame grabber does a half resolution acquire of the even field of the image.

caq_next: the frame grabber does a half resolution acquire, acquiring the next field, regardless of whether that field is even or odd.

FUNCTION: **caq_get_format_name()**

PURPOSE: Return a pointer to the most recently specified general format. (4400 only.)

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *clf.h*

USE:
```
char *caq_get_format_name()
```

NOTES: **caq_get_format_name()** returns a pointer to a string describing the general format most recently specified by **caq_specify_general_format()**. If a general format has not been specified, **caq_get_format_name()** returns NULL. The general format need not be currently loaded through a call to **caq_init()**.

VARIABLE: **caq_gplane**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
cip_buffer *caq_gplane;
```

NOTES: *caq_gplane* points to a **cip_buffer** representing the 3000/4000/5000 graphics buffer (the 2000 has no graphics buffers). The graphics buffer corresponds to the hardware graphics plane on the 3000, 4100/4200, and 5000, and to an image buffer used for graphics on the 4400. *caq_gplane* is guaranteed to use the same coordinates as the current *caq_image*.

On the 3000 only, *caq_gplane* has an aliased RAT if and only if the current *caq_image* is full resolution in y. The *CIP_FLAGS_ALLIASED_RAT* flags bit is set if and only if *caq_mode->y_res* is set to 0 or 1 (full resolution).

Also on the 3000 only, *caq_gplane* always has the *CIP_FLAGS_NO_Y_DOUBLING* bit for the **cip_buffer** set, indicating that no y pixel doubling is needed.

On the 3000/4000/5000, *caq_gplane* changes when **caq_resolution()** or **caq_graphics_select()** is called.

FUNCTION: **caq_graphics()**

PURPOSE: Put the frame grabber in graphics mode.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_graphics()
```

NOTES: When you call **caq_graphics()**, the monitor shows real-time video or a constant grey value with the contents of the frame grabber as a graphics overlay. On the 2000 the bottom two bits of *caq_image* are used as the source for the graphics: the low order two bits of the frame grabber contents are interpreted not as grey levels, but as one of the four graphics values: *CAQ_BLACK, CAQ_WHITE, CAQ_BLINK,* or *CAQ_PASS* (to pass video). The other bits are ignored.

The 3000/4000/5000 have an additional option specified by the value of *caq_mode->gplane_enable*. If this value is zero, these machines behave like a 2000. If this value is one, the source of the graphics is the special **cip_buffer** pointed to by *caq_gplane*, described earlier.

caq_graphics() throws the following errors:

- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode (2000 only)

FUNCTION: **caq_graphics_buffers()**

PURPOSE: **caq_graphics_buffers()** returns the number of graphics buffers supported in the given format and resolution.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
int caq_graphics_buffers(format,x_res,y_res)
   caq_format_type format;
   int x_res, y_res;
```

NOTES: **caq_graphics_buffers()** returns the number of graphics buffers supported in the given format and resolution. This function is for use on the 3000/4000/5000 only.

format: specifies the image format. It can have the values: *caq_format_rs170_576*, *caq_format_rs170_512*, *caq_format_rs170_wide*, *caq_format_pal*, or *caq_format_pal_512*.

x_res: the resolution in the x dimension. Setting *x_res* to 0 or 1 indicates full resolution in the x dimension. Setting *x_res* to 2 indicates half resolution in the x dimension.

y_res: the resolution in the y dimension. Setting *y_res* to 0 or 1 indicates full resolution in the y dimension. Setting *y_res* to 2 indicates half resolution in the y dimension. On the 4000/5000, *y_res* and *x_res* must be set to the same resolution.

FUNCTION: **caq_graphics_color()**

PURPOSE: Reassign the standard graphic values *CAQ_WHITE*, *CAQ_BLACK*, and *CAQ_BLINK* to the colors you specify (3000 only).

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

GLOBALS: *caq_blink_enable*

USE:

```
void caq_graphics_color(gval, red, green, blue, table)
   int gval, red, green, blue;
   char *table;
```

NOTES: **caq_graphics_color()** changes the output display table to make the graphics mode or the overlay mode display colors rather than black, white, and blinking black/white.

gval: the graphics value to reassign: *CAQ_WHITE*, *CAQ_BLACK*, or *CAQ_BLINK*.

red, *green*, and *blue:* components of the new color. Each argument can have a value from 0-255, where 0 indicates that the color is not present, and 255 is the maximum color saturation.

table: indicates the output display table to which the color assignments should be made. Currently, only the internal display table can be modified, so this argument must be set to 0.

The global variable *caq_blink_enable* defines whether the color assigned to *CAQ_BLINK* is displayed simply as that color, or whether colors blink between the color assigned to *CAQ_BLINK* and the color assigned to *CAQ_BLACK*. If *caq_blink_enable == 0*, then no blinking occurs and all pixels of graphics value *CAQ_BLINK* are displayed in the color assigned to *CAQ_BLINK*. If *caq_blink_enable == 1*, then all *CAQ_BLINK* pixels blink between the value of *CAQ_BLINK* and the value of *CAQ_BLACK*.

The default mapping of graphic values are as follows:

```
caq_graphics_color(CAQ_WHITE,255,255,255,0);

caq_graphics_color(CAQ_BLACK,0,0,0,0);

caq_graphics_color(CAQ_BLINK,0,0,0,0);

caq_blink_enable = 1;
```

FUNCTION: **caq_graphics_select()**

PURPOSE: Change the graphics buffer.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_graphics_select(index)
   int index;
```

NOTES: **caq_graphics_select()** selects a graphics buffer, and changes *caq_mode->graphics_buffer* to point to it.

index: the number of the buffer you want to use. Buffers are numbered from 0. On the 4400, this is the number of the image buffer to use for graphics. On the 3000, 4100/4200, and 5000, this is a buffer in the dedicated graphics plane. This number must be less than the number of buffers available for the format and resolution chosen.

If *index* is larger than the available number of buffers, the error *CAQ_ERR_FORMAT* is thrown. **caq_graphics_buffers()** may be used to determine the number of graphics buffers available in the format and resolution chosen.

caq_graphics_select() automatically updates the *caq_gplane* and *caq_rgplane* pointers to point to appropriate buffers.

This function is for use with the 3000/4000/5000 only.

MACRO: **CAQ_GROUP()**

PURPOSE: For the 4400 only, covert a system buffer number to a group buffer number.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
#define CAQ_GROUP0(x) (x*4+0)

#define CAQ_GROUP1(x) (x*4+1)

#define CAQ_GROUP2(x) (x*4+2)

#define CAQ_GROUP3(x) (x*4+3)
```

NOTES: These four macros are provided to help you select graphics buffers for specific image buffers when using half resolution operation on the 4400. Each macro is in the form *CAQ_GROUPx(y)*, where *x* is the group number (0 - 3), and *y* is the half resolution buffer number (0 - 3). The half resolution buffer number is specified relative to the full resolution buffer. Each macro returns a buffer number relative to the system numbering scheme (0 - 15).

VARIABLE: **caq_image**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
cip_buffer *caq_image;
```

NOTES: *caq_image* points to a special **cip_buffer** corresponding to the frame store. This image is one of the possible frame memory configurations for a given image format. It is pointed to from a table that is created by the function **caq_init()**.

When you call **caq_image_select()**, *caq_image*, *caq_even*, and *caq_odd* are changed to point to appropriate buffers. When you call **caq_graphics_select()** (available only on the 3000/4000/5000), *caq_gplane* and *caq_rgplane* are changed to point to appropriate

The tables are initialized by **caq_init()**.

FUNCTION: **caq_image_buffers()**

PURPOSE: Return the number of image buffers supported according to the given format and resolution.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
int caq_image_buffers(format,x_res,y_res)
   caq_format_type format;
   int x_res, y_res;
```

NOTES: *format:* specifies the image format.

x_res: the image resolution in the x dimension. Setting *x_res* to 0 or 1 indicates full resolution in the x dimension. Setting *x_res* to 2 indicates half resolution in the x dimension.

y_res: the image resolution in the y dimension. Setting *y_res* to 0 or 1 indicates full resolution in the y dimension. Setting *y_res* to 2 indicates half resolution in the y dimension. On the 4000/5000, *y_res* and *x_res* must be set to the same resolution.

caq_image_buffers() throws the following error:

- *CGEN_ERR_BADARG* if this function is called on the 2000 and a format is specified that is available only on the 3000/4000/5000

FUNCTION: **caq_image_select()**

PURPOSE: Change the *image_buffer* component of **caq_context**.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_image_select(index)
   unsigned index;
```

NOTES: **caq_image_select()** selects an image buffer and changes *caq_mod->image_buffer* to point to it.

```
void caq_image_select(index)
   unsigned int index;
```

index: the number of the buffer you want to use. *index* must be less than the number of buffers available for the format and resolution chosen. Buffers are numbered from 0. If *index* is larger than the available number of buffers, the error *CAQ_ERR_FORMAT* is thrown. You may call **caq_image_buffers()**, described above, to determine the number of image buffers available.

caq_image_select() automatically updates the *caq_image*, *caq_even*, and *caq_odd* pointers so each points to an appropriate buffer.

On the 2000 only, **caq_image_select()** throws the following errors:

- *CAQ_ERR_MOVING* if the moving acquisition state is not *off*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode

FUNCTION: **caq_init()**

PURPOSE: Initialize the frame grabber.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

GLOBALS: *caq_mode, caq_startup_context, caq_field, caq_sync_out, caq_sync_src, caq_format*

USE:

```
void caq_init()
```

NOTES: **caq_init()** creates a **cip_buffer** table based on the image format specified by *caq_format*. All windows and RATs relative to the frame store are invalid after a call to **caq_init()**.

On a 2000, **caq_init()** sets the contents of the graphics and image buffer to *CAQ_PASS*.

On a 3000/4000/5000 with the frame grabber in either graphics or overlay mode, **caq_init()** does one of the following:

- if *caq_context->gplane_enable* is set to 1, **caq_init()** sets the contents of the graphics buffer to *CAQ_PASS*.
- if *caq_context->gplane_enable* is set to 0, **caq_init()** sets the contents of the image buffer to *CAQ_PASS*.

If the frame grabber is not in graphics or overlay mode, **caq_init()** acquires an image. The frame grabber is left in the mode it was in before the call to **caq_init()**. **caq_init()** sets the values of the **caq_context** components *image_buffer* and *graphics_buffer*: **caq_init()** always selects image buffer 0; on the 3000, 4100/4200, and 5000, **caq_init()** always selects graphics buffer 0; on the 4400, **caq_init()** always selects the last available buffer for graphics.

caq_init() must be called before changes to the global variables, *caq_format, caq_sync_out,* and **caq_sync_src** become effective. It is called automatically whenever a vision processor is booted. **caq_init()** initializes the frame grabber and starts vertical blanking.

Note: A call to **caq_init()** may leave the system unable to acquire while the VP and camera resynchronize. Your application program should wait 5 - 10 vertical blanking periods after calling **caq_init()**, depending on the type of camera you use.

When *caq_format* is set to *caq_format_general_softacq*, **caq_init()** selects the software frame buffers, identified via **clf_set_buffer()**, as the image buffers. The image buffer index for a particular buffer is the value returned by **clf_set_buffer()**. An index of 0 is reserved for the hardware frame buffer.

caq_init() sets **cip_buffer** flag *CIP_FLAGS_REGULAR* when appropriate.

caq_init() throws the following errors:

- *CAQ_ERR_MOVING* if the moving acquisition state is not *off*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode

FUNCTION: **caq_init_moving()**

PURPOSE: Set up a moving acquisition for use with **caq_acquire()** on trigger line 2.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:
```
void caq_init_moving(mode)
   caq_context *mode;
```

NOTES: *mode:* a pointer to an instance of **caq_context**. The variable *mode->trigger* must be 2.

Moving triggers support any of the 4 values of *mode->field*, but half resolution solid state cameras work properly only with *mode->field = caq_next.*

The lines *mode->cti_start[0]* to *mode->cti_end[0]* are the even field charge transfer interval during which the strobe will not flash.

The lines *mode->cti_start[1]* to *mode->cti_end[1]* are the odd field charge transfer interval during which the strobe will not flash.

Trigger 2 must be enabled by a call to **ctg_init()** before calling **caq_init_moving()**

throws the following errors:

- *CAQ_ERROR_MOVING* if the state is not *armed*, *triggered* or *processing*
- *CAQ_ERROR_ASYNC* if the system is in asynchronous mode

VARIABLE: **caq_line**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
short caq_line;
```

NOTES: In RS-170 and CCIR formats, the video system is always running, and *caq_line* contains the current line number.

On the 3000/4000/5000, **caq_line** is reset a few lines after the beginning of vertical sync. It is equal to the value *CAQ_EVEN_LINE* when the first line of the even field is being displayed, and to *CAQ_ODD_LINE* when the first line of the odd field is being displayed.

Note: This is NOT the standard RS-170 definition of line number

Since *caq_line* is a hardware location, it is a *read only* variable.

Normally you do not have to keep track of line numbers, except that the *cti_start* and *cti_end* values must be set to the Cognex-supplied values for a particular camera in order to do moving acquisitions.

If it is necessary to convert a *caq_image* y pixel coordinate to a line number use:

```
line = (y>>1) + ((y & 1) ? CAQ_ODD_LINE : CAQ_EVEN_LINE);
```

caq_line is used internally for timing and image acquisition synchronization.

Some 2000 applications use *caq_field* to detect the start of vertical blanking and *caq_line* to detect the end so that screen updates can be done during blanking. The 3000/4000/5000 allow simultaneous video and CPU access, so updates can be done at any time, making this computation unnecessary.

FUNCTION: **caq_load_vb_action()**

PURPOSE: Load the graphics action routine (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE: `void caq_load_vb_action()`

NOTES: **caq_load_vb_action()** loads the graphics action routine, which executes at vertical blanking. The action routine allows full graphics capability on frozen images without using the graphics plane *caq_gplane*.

VARIABLE: **caq_mode**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:
```
caq_context *caq_mode;
```

NOTES: *caq_mode* points to an instance of **caq_context** that controls the frame grabber (except during image acquisition). The frame grabber hardware is updated from this structure during the vertical blanking period every 1/60 second (1/50 second on the 3000/4000/5000 in CCIR mode).

To enable or disable the graphics plane of the system, you must use **caq_set_gplane_enable()** to set the value of *caq_mode->gplane_enable*. You should not set *caq_mode->gplane_enable* directly.

The default value for *caq_mode* after booting a volume image is a pointer to the variable *caq_startup_context*.

When you switch cameras by resetting *caq_mode->camera*, you should wait at least 3 field times (~50 msec) to allow the VP and camera to resynchronize.

VARIABLE: **caq_odd**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
cip_buffer *caq_odd;
```

NOTES: *caq_odd* points to the **cip_buffer** corresponding to the odd field of the frame store image.

The pixel at coordinates (x,y) in *caq_odd* is the same as that at (x,2y) in *caq_image*.

VARIABLE: **caq_odd_line**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:
```
int caq_odd_line;
```

NOTES: *caq_odd_line* holds the number of the first line of *caq_odd*. *caq_odd_line* is initialized by **caq_init()** based on the specified image format.

FUNCTION: **caq_overlay()**

PURPOSE: Display a stored image with a graphic overlay.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_overlay()
```

NOTES: **caq_overlay()**, available only on the 3000/4000/5000, displays the stored image with a graphic overlay. The image comes from *caq_image* and the overlay from *caq_gplane*. Before you call **caq_overlay()** you should set *caq_mode->gplane_enable* to 1.

On the 3000, if *caq_stored_eightbit* is nonzero, the output displays 64 grey levels with graphics. If the stored image is a six bit image, the output can display the grey scale quantized to only 16 levels plus graphics.

On the 4000/5000, if *caq_stored_eightbit* is nonzero, the output displays 256 grey levels with graphics. If the stored image is a six bit image, the output can display the grey scale quantized to only 64 levels plus graphics.

caq_overlay() throws the following errors:

- *CAQ_ERR_MOVING* if the moving acquisition state is not *off*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode (2000 only)

FUNCTION: **caq_ready()**

PURPOSE: Determine if a call to **caq_acquire()** will return in a bounded amount of time.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
bool caq_ready(mode)
   caq_context *mode;
```

NOTES: **caq_ready()** returns TRUE if a call to **caq_acquire()** will not "hang," and returns FALSE if a moving acquisition or acquisition-after-move is specified and the corresponding trigger event has not yet occurred.

mode: points to the **caq_context** to be used by **caq_acquire()**.

caq_ready() throws the following errors:

- *CAQ_ERR_MOVING* if the call to **caq_acquire()** would throw that error
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode

FUNCTION: **caq_realtime()**

PURPOSE: Put the frame grabber in real-time mode.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_realtime()
```

NOTES: **caq_realtime()** puts the frame grabber into real-time mode on the 3000/4000/5000 (not defined on the 2000). It displays either real-time input or a constant, as selected by *caq_mode->display_value*.

caq_realtime() throws the following errors:

- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode (2000 only)

FUNCTION: **caq_reset_vb_action()**

PURPOSE: Disable the graphics action routine (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:
```
void caq_reset_vb_action()
```

NOTES: **caq_reset_vb_action()** disables the graphics action routine previously enabled using the function **caq_load_vb_action()**.

FUNCTION: **caq_resolution()**

PURPOSE: Change the *x_res* and *y_res* components of a **caq_context** structure.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_resolution(x_res,y_res)
   int x_res, y_res;
```

NOTES: **caq_resolution()** changes the resolution of an acquisition.

x_res and *y_res:* the new resolution. A value of 0 or 1 for the *x_res* or *y_res* argument indicates full resolution in that dimension. A value of 2 for *x_res* or *y_res* indicates half resolution in that dimension. On the 4000/5000, *x_res* and *y_res* must be set to the same resolution.

caq_resolution(*x_res,y_res***)** sets *caq_mode->x_res* and *caq_mode->y_res* to the supplied values, and modifies the five global pointers *caq_image*, *caq_even*, *caq_odd*, *caq_gplane*, and *caq_rgplane*, as appropriate for the new values. Because the number of image buffers and graphics buffers allowed changes when the resolution changes, **caq_resolution()** resets *caq_mode->graphics_buffer* and *caq_mode->image_buffer*. **caq_resolution()** always selects image buffer 0; on the 3000, 4100/4200, and 5000, **caq_resolution()** always selects graphics buffer 0; on the 4400, **caq_resolution()** always selects the last available buffer for graphics.

On the 2000 only, **caq_resolution()** throws the following errors:

- *CAQ_ERR_MOVING* if the moving acquisition state is not *off*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode

On the 4000/5000 only, **caq_resolution()** throws the following error:

- *CAQ_ERR_FORMAT* if *x_res* and *y_res* are not set to the same resolution

VARIABLE: **caq_rgplane**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
cip_buffer *caq_rgplane;
```

NOTES: *caq_rgplane* points to one of the **cip_buffers** representing the 3000 graphics plane. This **cip_buffer** is the raw graphics plane and is always half resolution in y.

caq_rgplane never has an aliased RAT. It always has the *CIP_FLAGS_NO_Y_DOUBLING* bit set, indicating that no y pixel doubling is needed.

caq_rgplane changes when either **caq_resolution()** or **caq_graphics_select()** is called.

caq_rgplane can be used for speed critical applications by functions that use absolute coordinates instead of *caq_image* coordinates or the coordinates of any window of *caq_image*.

FUNCTION: **caq_set_gplane_enable()**

PURPOSE: Enables or disables the graphics plane of the system.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_set_gplane_enable (gplane_flag)
   bool gplane_flag;
```

caq_set_gplane_enable() sets the value of *caq_mode->gplane_enable* to enable or disable the graphics plane of the system. You should not set the value of *caq_mode->gplane_enable* directly.

gplane_flag, if set to TRUE, enables the graphics plane of the system; if set to FALSE, it disables the graphics plane of the system.

FUNCTION: **caq_specify_general_format()**

PURPOSE: On the 4400 only, specify a general format for a line-scan camera.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *clf.h*

USE:
```
void caq_specify_general_format (format)
   caq_general_format *format;
```

NOTES: **caq_specify_general_format()** specifies video timing information to be loaded when you have specified a **caq_format_type** of *caq_format_general* or *caq_format_general_softacq*.

caq_general_format is a proprietary data structure, and its contents are not public; instead, Cognex provides predefined names for *format*:

format is a pointer to a **caq_general_format** data structure which describes the timing information for the camera you use (such as whether to use TTL or RS-422 operation).

For the Dalsa CL-E1-2048 camera:

cls_cle1_2048_512_format for 512 x 512 acquisition

cls_cle1_2048_1kx512_format for 1024 x 512 acquisition

cls_cle1_2048_1k_format for 1024 x 1024 acquisition

cls_cle1_2048_2k_format for 2048 x 2048 acquisition

For the EG&G Reticon LC1902 camera:

cls_lc1902_512_format for 512 x 512 acquisition

cls_lc1902_1kx512_format for 1024 x 512 acquisition

cls_lc1902_1k_format for 1024 x 1024 acquisition

cls_lc1902_2k_format for 2048 x 2048 acquisition

For the Nippon Electronic Devices YD2048CH camera:

cls_nd_512_format for 512 x 512 acquisition

cls_nd_1kx512_format for 1024 x 512 acquisition

cls_nd_1k_format for 1024 x 1024 acquisition

cls_nd_2k_format for 2048 x 2048 acquisition

For the Texas Instruments MC-1000PU-20 or WU-20 cameras:

clf_ti_format for 1024 x 1024 acquisition

clf_ti_512_format for 1024 x 512 acquisition

For the Texas Instruments MC-1000PU-20 or WU-20 cameras:

clf_ti_format for 1024 x 1024 acquisition

clf_ti_512_format for 1024 x 512 acquisition

For large-format image display:

clf_rs170_format for display of large-format (non-interlaced) images on an RS-170 monitor. It is equivalent to *caq_format_rs170_512*, except that when you call **caq_init()** to put it into effect, no acquisition is done. *clf_rs170_format* offers the advantage of reduced display flicker when switching video formats.

Note that the 512 x 512 format allows up to four image buffers, and the 1024 x 512 format allows two. Specify the image buffer you want in the *image_buffer* field of the **caq_context** structure that you pass to **caq_acquire()**.

VARIABLE: **caq_startup_context**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
caq_context caq_startup_context;
```

NOTES: By default, after booting a volume image *caq_mode* points to *caq_startup_context*.

```
caq_context caq_startup_context =
{
   CAQ_STD_GAIN,      /* default gain */
   CAQ_STD_OFFSET,    /* default offset */
   0,          /* no trigger for stationary acquisition */
   2,          /* settling time, 1/30 second */
   0,          /* no strobe */
   1,          /* camera 1 */
   1,          /* acquire polarity positive image */
   1,          /* display polarity positive image */
   -1,         /* display raw video */
   caq_full,/* full field acquire */
   0,          /* no split */
   {CAQ_CTI_START_EVEN,CAQ_CTI_START_ODD},
               /* starting line nos. of charge xfer */
   {CAQ_CTI_END_EVEN,CAQ_CTI_END_ODD},
               /* ending line numbers of charge xfer */
   1,          /* full resolution in x */
   1,          /* full resolution in y */
   0,          /* image buffer to use */
   0,          /* graphics buffer to use */
   0,          /* eight bit frame store disabled */
   0,          /* graphics plane disabled */
   0,          /* input control - not supported */
   0,          /* display control - not supported */
   CAQ_STATE_INACTIVE,/* state for async acquisition */
   0,          /* pointer to next caq_context structure */
   0,          /* active time 1 */
   0,          /* active time 2 */
   0,          /* acquire overrun count */
   0,          /* trigger overrun count */
   0,          /* camera-specific parameters */
   0           /* region of interest parameters */
};
```

Note: On a 4400 the graphics buffer is set to the last available image buffer.

The following defined constants are available for use with the **caq_context** structure. Their values are not shown because they may differ among the vision processors.

```
CAQ_STD_GAIN   /* startup default value */
CAQ_STD_OFFSET /* startup default value */
CAQ_GRAPHICS   /* mask for the graphics bits */
CAQ_WHITE      /* graphics value for white */
CAQ_PASS       /* graphics value for pass video */
CAQ_BLINK      /* graphics value for blink */
CAQ_BLACK      /* graphics value for black */
```

The following macros are also defined:

```
#define CAQ_EVEN_LINE caq_even_line
#define CAQ_ODD_LINE caq_odd_line
#define CAQ_WIDTH caq_image->width
#define CAQ_HEIGHT caq_image->height
```

VARIABLE: **caq_stored_eightbit**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
bool caq_stored_eightbit;
```

NOTES: *caq_stored_eightbit* controls the effective depth of the 3000/4000/5000 during display mode or overlay mode (*caq_stored_eightbit* does not exist on a 2000). It may have values 0 (stored image is 6 bit) or 1 (stored image is 8 bit). It is set during an acquisition according to the value of the *eightbit_enable* field of the **caq_context** structure used during acquisition.

When *eightbit_enable* is 0, the low six bits of the frame memory are used for output intensity and the two most significant bits are ignored. When *eightbit_enable* is 1, the six most significant bits are used and the low two bits are ignored.

The default value after booting a volume image for *caq_stored_eightbit* is 0. You should not modify this variable if you want your software to run in 2000 compatibility mode.

VARIABLE: **caq_stored_field**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
enum caq_frame caq_stored_field;
```

NOTES: *caq_stored_field* indicates which video field was last acquired. This variable is set when an acquisition completes.

caq_stored_field is a variable of type **caq_frame**. It can be one of the following:

caq_full: both fields acquired

caq_even_field: the even field was acquired

caq_odd_field: the odd field was acquired

When doing a half resolution operation of type *caq_next* from a full resolution camera, *caq_stored_field* can be used to determine whether to add a half pixel offset in *y* to a computed position to compensate for the difference in position between the two fields. To demonstrate this, compare the positions computed for two acquisitions of the exact same scene, one with *field == caq_even* and one with *field == caq_odd*.

The value of the global variable *caq_stored_field* indicates which video field was last acquired. This variable is set when an acquisition completes.

VARIABLE: **caq_stored_image_polarity**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
bool caq_stored_image_polarity;
```

NOTES: *caq_stored_image_polarity*, together with *caq_mode->display_polarity*, governs the translation of pixel values in the frame grabber to the output video signal when the frame grabber is in display mode.

If *caq_stored_image_polarity* and *caq_mode->display_polarity* are the same, the pixel value 0 is mapped to black, and the pixel value 63 (255 on the 3000/4400 in eight-bit mode) maps to white. If these two globals are not equal then the inverse mapping occurs.

Acquiring an image sets *caq_stored_image_polarity* to the value *caq_mode->acquire_polarity*.

Normally the only time you will modify *caq_stored_image_polarity* is when writing a stored image into the frame grabber for display. If the copied image is "normal" (0 displays as black) set *caq_stored_image_polarity* to 1; otherwise, set *caq_stored_image_polarity* to 0.

VARIABLE: **caq_sync_out**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
enum caq_sync caq_sync_out;
```

NOTES: **caq_sync_out** specifies the signal sent to the camera as either separate vertical and horizontal sync, composite sync, or no sync (3000/4000/5000 only). *caq_sync_out* is of type **caq_sync**:

```
typedef enum caq_sync{
   caq_separate,
   caq_composite,
   caq_nosync
}caq_sync;
```

The hardware provides two sync lines: vertical sync and a line that can be switched under software control to either horizontal or composite sync. When *caq_sync_out* is set to *caq_separate*, both lines are used. Typically, only when *caq_sync_out* is set to *caq_composite*, can the software set line be hooked up to the camera.

When **caq_sync_out** is set to *caq_nosync*, the sync outputs are disabled and sync may be provided externally; this feature is available only on the 3000 and 4400. On the 4100/4200 and 5000, selecting *caq_nosync* has no effect. The only way to shut off sync generation on the 4100/4200 and 5000 is to set the variable **caq_sync_src** to *caq_external*.

Any change to this variable takes effect on the next call to **caq_init()**.

The Panasonic WV-CD50 and Sony XC-38 cameras require separate horizontal and vertical drive. The Sony XC-57 and XC-77 cameras accept either composite sync or separate horizontal and vertical sync.

After booting a volume image, the default value for **caq_sync_out** is *caq_separate*.

VARIABLE: **caq_sync_src**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:
```
enum caq_sync_src_type caq_sync_src;
```

NOTES: Specifies the camera sync as internal (coming from the frame grabber) or external (coming from the sync signal present in the video input). External sync is available only on the 3000/4000/5000. The variable is of type **caq_sync_src_type**:

```
typedef enum{
   caq_internal,
   caq_external
}caq_sync_src_type;
```

When **caq_sync_src** is set to *caq_internal*, the frame grabber provides the camera sync.

When **caq_sync_src** is set to *caq_external*, the video system is phase-locked to the sync signal present in the video input. When *caq_external* is selected, *caq_sync_out* should be set to *caq_nosync* because the board and the source cannot lock to each other. On the 4100/4200 and 5000, selecting *external_sync* shuts off the sync generator regardless of the setting of the variable **caq_sync_out**.

Any change to this variable takes effect on the next call to **caq_init()**.

After booting a volume image, the default value for **caq_sync_src** is *caq_internal*.

FUNCTION: **caq_video()**

PURPOSE: Put the frame grabber in display, real-time, graphics, or overlay mode, whichever was the last selected.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

USE:

```
void caq_video()
```

NOTES: **caq_video()** puts the frame grabber into display mode, graphics mode or overlay mode, whichever was the last selected. Calling **caq_video()** immediately after booting a 2000, 4000, or 5000 vision processor puts the frame grabber in graphics mode. Calling **caq_video()** immediately after booting a 3000 puts the frame grabber in overlay mode.

caq_video() throws the following errors:

- *CAQ_ERR_MOVING* if the system is *armed* or *triggered*
- *CAQ_ERR_ASYNC* if the system is in asynchronous mode (2000 only)

VARIABLE: **caq_video_mode**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *caq_def.h*

DEFINITION:

```
char caq_video_mode;
```

NOTES: *caq_video_mode* controls how the 3000/4000/5000 behaves when in CPU mode. When *caq_video_mode* is set to zero, the 3000/4000/5000 behave like the 2000 when in CPU mode. If it is set to one, the 3000/4000/5000 ignore CPU mode and leaves the display unchanged from the previous mode. Since **caq_acquire()** puts the software in CPU mode, you can set this variable to keep acquisitions from changing the display mode.

The default value after booting a volume image for *caq_video_mode* is 0.

caq_video_mode is not available on a Cognex 2000.

FUNCTION: **cblob_connectivity()**

PURPOSE: Perform blob analysis on connected regions of an image.

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

USE:

```
cblob_results *cblob_connectivity(cblob_function,
      image,params,results,rip_pp)
   cblob_results *(*cblob_function)();
   cip_buffer *image;
   cblob_params *params;
   cblob_results *results;
   crip_buffer **rip_pp;
```

NOTES: **cblob_connectivity()** returns a pointer to a **cblob_results** structure.

cblob_function: the address of the desired blob analysis function; i.e., **cblob_zero()**, **cblob_one()**, or **cblob_two()**. If this parameter is set to 0, connectivity analysis will be run but no blob analysis will be performed.

image: points to the image on which to run the analysis.

params: points to the record specifying the parameters of the analysis.

results: points to an array of results records to be filled in. If *results=0*, the function allocates a number of results records on the heap equal to the number of results records requested in the **cblob_params** structure. You must deallocate this space. If results is nonzero, the function takes it to be the address of an array of records you have allocated. In either case, the function returns a pointer to the first results record.

rip_pp: if nonzero, points to a location where **cblob_connectivity()** will return a pointer to a **crip_buffer** that describes the regions and runs found. If *rip_pp* is set to 0, this information is not returned. If *rip_pp* is nonzero, the caller is responsible for deallocating the connectivity results; **crip_delete()**, is the proper function for this purpose.

FUNCTION: **cblob_one()**

PURPOSE: Compute the area, center of mass, extrema, and selected second moments of inertia.

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

USE:

```
cblob_results *cblob_one(image,params,results)
   cip_buffer *image;
   cblob_params *params;
   cblob_results *results;
```

NOTES: **cblob_one()** returns a pointer to an array of **cblob_results** structures.

image: points to the image on which to operate.

params: points to the parameters structure.

results: points to an array of one or more results records. If *results* is NULL, the function allocates a number of results records requested in the **cblob_params** structure; for whole image analysis only one record is filled in.

On the 2000, execution time is 0.6 µs. per pixel plus ~5 ms. overhead for arbitrary and absolute maps; 0.9 µs. per pixel plus ~6 ms. overhead for relative maps.

STRUCTURE: **cblob_params**

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

DEFINITION:

```
typedef struct{
   cmap_function wf;
   cmap_map_data mp;
   short results;
   double min_area;
   double xmin,ymin,xmax,ymax;
   int flags;
   int reserved[3];
}cblob_params;
```

NOTES: **cblob_params** is the blob analysis parameters structure.

wf: specifies the type of pixel weighting function (or map) to be used in blob analysis. *wf* may have values of: *cmap_arbitrary* (for an arbitrary map), *cmap_absolute* (for an absolute map), *cmap_relative* (for a relative map), *cmap_binary* (for a binary map) and *cmap_identity* (for an identity map).

In previous releases of the software, this component was of type **cblob_maps** and the values *cmap_binary* and *cmap_identity* did not exist. **cblob_maps** has been **#defined** to be equivalent to type **cmap_function**. Also, the enumeration values of three of the map types have been **#defined** as follows:

```
#define cblob_arbitrary cmap_arbitrary
#define cblob_absolute cmap_absolute
#define cblob_relative cmap_relative
```

Any code you have written using the **cblob_*** data types will be compatible with this release.

mp: a union of data structures, each of which defines the parameters of one of four types of maps: arbitrary, absolute, relative or binary (there are no parameters defined for an identity map). The format of data stored in this union (i.e., the active component of the union) is chosen based on structure element *wf*.

In previous releases of the software, the union *mp* was not a **typedef** (**cmap_map_data**). This does not affect any structure references; any code you have written in which *mp* is not a **typedef** will be compatible with this release.

results: specifies the number of results records you have provided (allocated); hence, the maximum number of results records to be returned by the blob analysis routines. Results records are numbered (*id*) consecutively starting at 1 and are stored in an array.

If the value of *results* is greater than 0, that number of results records are allocated; however, only 1 results structure will be filled in. If *results=0*, the error *CGEN_ERR_BADARG* is thrown. For whole image blob analysis, *results* should be set to 1.

min_area: for connected blob analysis, *min_area* specifies the minimum total pixel weight (connected pixels) that is to be considered a blob. Any object having less area is not reported; i.e., no results record for it is returned. The purpose of *min_area* is to reduce the impact of noise or extraneous marks on the analysis of the scene.

xmin, ymin, xmax, ymax: specify the weight (area in pixels) to exclude when computing blob extrema, the coordinates to which a blob extends. This improves repeatability of blob measurements in the presence of noise or spatial quantization errors.

flags: specifies whether or not euler number and perimeter values will be returned in the **cblob_results** structure by the functions **cblob_zero()**, **cblob_one()** and **cblob_two()**. Set flags to cblob_perim_flag to cause euler and perimeter values to be returned. When flags is set to *cblob_perim_flag*, the stroke width also is returned in the results structure.

reserved: reserved for future use.

FUNCTION: **cblob_regions()**

PURPOSE: Runs blob analysis on selected pixels in an image. The pixels are specified by connectivity data obtained from the **cblob_connectivity()**.

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

USE:

```
cblob_results *blob_regions(cblob_function,image,
   params,rip,results)
   cblob_results *(*cblob_function)();
   cip_buffer *image;
   cblob_params *params;
   crip_buffer *rip;
   cblob_results *results;
```

NOTES: **cblob_regions()** returns a pointer to an array of **cblob_results** structures. **cblob_regions()** allows you to separate the connectivity analysis from the other blob analysis measurements.

cblob_function: points to the blob analysis function to be run on the specified regions. *cblob_function* should be set to *cblob_zero*, *cblob_one* or *cblob_two*

image: points to the image containing the regions to be analyzed. *ip* must be large enough to accommodate the regions specified by the connectivity data.

params: points to the **cblob_params** record specifying the analysis parameters.

rip: points to the **crip_buffer** containing the connectivity data. This **crip_buffer** is returned by the function **crip_connectivity()**.

results: points to an array of results records to fill. If *results* is 0, **cblob_regions()** allocates on the heap the number of results records requested in the **cblob_params** structure. You must deallocate this space. If *results* is nonzero, **cblob_regions()** uses it as the address of an allocated array of records. In either case, the function returns a pointer to the first results record. The number of results records containing valid data is limited by the number of available regions which meet the minimum area specified in the **cblob_params** structure. The *min_area* parameter is compared to *n* in the associated **crip_region** data structure to determine whether or not to compute blob analysis. The weighted area is computed and returned for eligible regions. The first unused result record has *id = 0*.

cblob_regions() throws the following errors:

- *CGEN_ERR_BADARG* if *cblob_function*, *image*, *params*, or *rip* is NULL
- *CBLOB_ERR_PARAMS* if the number of results specified in the **cblob_params** structure is 0

STRUCTURE: **cblob_results**

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

DEFINITION:

```
typedef struct{
   short id;
   int time;
   short
      left_tail,right_tail,zero,one;
   double
      area,
      cmx,cmy,mxx,myy,mxy;
   double
      xmin,ymin,xmax,ymax,medx,medy;
   double
      pmi1,pmi2,
      angle;
   double perimeter;
   short euler;
   double
      strwd,strln;
   int reserved[4];
}cblob_results;
```

NOTES: **cblob_results** is the blob analysis results structure.

id: the blob id number. Each blob found has its own id number. For whole image blob analysis, *id=1*. For connected blob analysis, these records are numbered consecutively starting at *id* number 1, in order of decreasing area. Results records are stored in an array. The first *id=0* marks the end of the array of results.

time: the execution time of the called blob analysis function.

For **cblob_connectivity()**, which returns multiple results records, the *time* returned in the first record includes both the time to do the connectivity analysis and the time to perform blob analysis. For subsequent records, *time* does not include the connectivity analysis; it is only the time used for making the other measurements on the blob. For **cblob_regions()**, *time* is only the time to perform blob analysis.

left_tail: the left tail of the intensity histogram according to the type of map specified in the parameters record.

- If the parameters record specifies an identity map, *left_tail* = 0.
- If the parameters record specifies an arbitrary map, *left _tail* = 0.
- If the parameters record specifies an absolute map, *left_tail* = 0.

- If the parameters record specifies a relative linear map, *left_tail* specifies the computed left tail (grey value) of the intensity histogram.
- If the parameters record specifies a binary map, *left_tail* specifies the computed left tail (grey value) of the intensity histogram.

right_tail: the right tail of the intensity histogram according to the type of map specified in the parameters record.

- If the parameters record specifies an identity map, *right_ tail* = 255.
- If the parameters record specifies an arbitrary map, *right_tail* = 255.
- If the parameters record specifies an absolute map, *right_ tail* = 255.
- If the parameters record specifies a relative linear map, *right_tail* specifies the computed right tail (grey value) of the intensity histogram.
- If the parameters record specifies a binary map, *right_tail* specifies the computed right tail (grey value) of the intensity histogram.

zero and *one:* contain the grey values assigned weights 0.0 and 1.0 respectively.

- If the parameters record specifies an identity map, *zero* = 0 and *one* = 255.
- If the parameters record specifies an arbitrary map, *zero* = 0 and *one* = the value specified in *cblob_params.mp.ar.w*.
- If the parameters record specifies an absolute map, *zero* = the value specified in *cblob_params.mp.ab.zero* and *one* = the value specified in *cblob_params.mp.ab.one*.
- If the parameters record specifies a relative linear map, then *zero* and *one* contain the computed *zero* and *one* grey values.
- If the parameters record specifies a binary map, both *zero* contains the computed (grey value) threshold.

area: the total weight of the blob. Since pixels are weighted, this measurement is not equivalent to an area measurement computed by binarizing the image and counting all the object pixels. The area measurement may have a fractional component.

cmx and *cmy:* are the coordinates of the center of mass with respect to the upper corner of the image. Computed by functions **cblob_one()** and **cblob_two()**.

mxx, myy and *mxy:* the second moments about the axes passing through the center of mass and parallel to the x and y axes. *mxy* is the moment about

the diagonal. *mxx* and *myy* are computed by function **cblob_one()** and **cblob_two()**. *mxy* is computed by function **cblob_two()**.

xmin, xmax, ymin and *ymax:* are the blob extrema (the coordinates to which a blob extends). They are computed by functions **cblob_one()** and **cblob_two()**.

These components may hold fractional values. For example, the values (*xmin,ymin*) = (*.5,.5*) would indicate the center of pixel (0,0).

medx and *medy:* the median point of the blob.

The median point of a blob is the point such that the total weight to the left and right of the point are equal and the total weight above and below the point are equal. medx and medy are computed by function **cblob_one()** and **cblob_two()**.

pmi1 and *pmi2:* the principal moments of inertia about the principal axes. The ratio of the second to the first principal moment is a measure of blob elongation. If the principal axes do not exist, the principal moments are equal and indicate the moment of inertia about any line through the center of mass. The principal moments are computed by function **cblob_two()**.

angle: the angle of the first principal axis measured counterclockwise with respect to the x axis, in degrees. *angle* has the range -90.0° to +90.0°.

perimeter: the perimeter of the blob in pixels.

The value of *perimeter* is determined by counting the number of instances in a thresholded version of the image where an object pixel (pixel with value 1) is adjacent to a background pixel (pixel with value 0). These pixels may be adjacent either horizontally or vertically.

The returned perimeter value is really an "estimated" value, and in many cases will be larger than the expected value because what is actually measured in the image is the jagged edge of a thresholded object, not the length of a smooth curve. For a diagonal line, for example, the returned value will be $\sqrt{2}$ times the actual value. On average, over all angles of inclination, the perimeter of an object is overestimated by:

$$\frac{4}{\pi} = 1.273$$

You can more closely approximate the actual perimeter of the object by dividing the returned value by this number.

euler: euler number, this is equal to the number of objects (blobs) minus the number of holes.

strwd: the stroke width of the blob. When the flags field of the **cblob_params** structure is set to *CBLOB_PERIM_FLAG*, the stroke width is returned in the results structure.

strln: reserved for future use.

FUNCTION: **cblob_two()**

PURPOSE: Compute all of the **cblob_one()** results plus *mxy*, *pmi1*, *pmi2*, and *angle*.

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

USE:

```
cblob_results *cblob_two(image,params,results)
   cip_buffer *image;
   cblob_params *params;
   cblob_results *results;
```

NOTES: **cblob_two()** returns a pointer to an array of **cblob_results** structures.

image: points to the image on which to operate.

params: points to the parameters structure.

results: points to an array of one or more results records. If *results* is NULL, the function allocates a number of results records requested in the **cblob_params** structure; for whole image analysis only one record is filled in.

On the 2000, execution time is 1.1 µs. per pixel plus ~9 ms. overhead for arbitrary and absolute maps; 1.4 µs. per pixel plus ~10 ms. overhead for relative maps.

FUNCTION: **cblob_zero()**

PURPOSE: Compute the area of a blob.

REFERENCE: Vision Tools: Inspection

INCLUDE FILE: *cblob.h*

USE:

```
cblob_results *cblob_zero(image,params,results)
   cip_buffer *image;
   cblob_params *params;
   cblob_results *results;
```

NOTES: **cblob_zero()** returns a pointer to an array of **cblob_results** structures.

image: points to the image on which to operate.

params: points to the parameters structure.

results: points to an array of one or more results records. If *results* is NULL, the function allocates a number of results records requested in the **cblob_params** structure; for whole image analysis only one record is filled in.

On the 2000, the execution time is 0.3 µs. per pixel plus ~1 ms. of overhead. The overhead time is slightly dependent on the choice of weighting function (slowest for relative maps), but the time per-pixel is not.

MACRO: **CBUS_ASSERT_RESET**

PURPOSE: A symbolic offset in shared memory that you can use to reboot a 4000-series vision processor. It is only available for VMEbus access.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_ASSERT_RESET
```

NOTES: To reboot a 4000`series vision processor over the VMEbus, your host should bitwise OR *CBUS_RESET_BIT* with the following location: *CBUS_SHARED_BASE_ADDRESS + CBUS_ASSERT_RESET*. This asserts a reset on the 4000.

After a reset is asserted, it must be de-asserted after the reset has had enough time to take hold. See the discussion of the macro **CBUS_DEASSERT_RESET** for details.

FUNCTION: **cbus_clear_isa_int()**

PURPOSE: Clears the pending ISA bus interrupt from the 5000 to the PC (5000 only).

REFERENCE: System Software: ISA Bus I/O

INCLUDE FILE: *cbus_isa.h*

USE:

```
int cbus_clear_isa_int();
```

NOTES: Only one interrupt request from the 5000 to the PC host can be pending. This function clears the interrupt request regardless of the line over which the request is pending.

If the operation is successful, this function returns 0.

FUNCTION: **cbus_clear_vme_int()**

PURPOSE: De-asserts an interrupt on the 4000. It is supplied for use in the action routine whose address you pass to the function **cbus_register_interrupter()**.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
void cbus_clear_vme_int(level)
   char level;
```

NOTES: *level:* the line (in the range 1 to 7) on which to de-assert the interrupt. *level* is ignored on the 4100/4200, and any outstanding interrupt is cleared.

Use **cbus_clear_vme_int()** if the interrupt handler has not responded to the interrupt, and the software on the 4000 wants to de-assert the interrupt request.

MACRO: **CBUS_DEASSERT_RESET**

PURPOSE: A symbolic offset in shared memory that you can use to de-assert a reset on a 4000-series vision processor. It is only available for VMEbus access.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:
```
#define CBUS_DEASSERT_RESET
```

NOTES: After a reset is asserted, it must be de-asserted after the reset has had enough time to take hold. To cause the appropriate delay, write any code on your VMEbus host that causes the device to wait at least 200 µs on a 4400 or 1 µs on a 4100/4200 before de-asserting the reset.

To de-assert the reset on a 4000, your host should bitwise AND the location *CBUS_SHARED_BASE_ADDRESS +CBUS_DEASSERT_RESET* with the value *~CBUS_RESET_BIT*.

MACRO: **CBUS_EXP_INT()**

PURPOSE: Executes a simulated expansion port interrupt on the 4000. Before using this macro, you must initialize the interrupt on the 4000 by calling the function **cio_expansion_int()**. This macro is called on the VMEbus host.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE: `CBUS_EXP_INT(base)`

NOTES: *base:* the host's base address for the 4000 in A16 or A0 space; of type: *pointer to character.*

MACRO: **CBUS_IPCR6**

PURPOSE: An offset to a register on the 4000 that indicates the 4000's current state. It can be written to only by the 4000; however, it can be read over the bus, using the symbolic location *CBUS_IPCR6*.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:
```
#define CBUS_IPCR6
```

NOTES: The states are represented by the following constants:

- *CBUS_VP_HALTED:* The 4000 has stopped running.
- *CBUS_LOCAL_RESET:* The 4000 has reset itself.
- *CBUS_VME_RESET:* The 4000 has been reset by a device on the VMEbus.
- *CBUS_VP_DIAG_RUN:* The 4000 is running start-up diagnostics.
- *CBUS_VP_DIAG_FAIL:* Start-up diagnostics have failed.
- *CBUS_VP_POLLING:* The 4000 is polling its boot devices, including the VMEbus, waiting for a device to supply boot data. Shared memory is initialized for booting only at this point.
- *CBUS_VP_RUNNING:* The 4000 is past the polling state. Either it is running your application or booting up. Once this is set, you can check the "magic address" *CBUS_MAGIC_ADDR* to see if shared memory is fully initialized. Although this flag is set by the operating system, it also can be reset by your application code on the 4000. You set this value by calling the function **cbus_set_state()**.
- *CBUS_APP_UP: IPCR6* can be set to this value by your application code on the 4000 to synchronize the host with an application on the 4000. It is intended to communicate that your application is running. You can set *IPCR6* to this value by calling the function **cbus_set_state()**.
- *CBUS_APP_DOWN: IPCR6* can be set to this value by your application code on the 4000 to synchronize the host to the application on the 4000. It is intended to communicate that your application is not running. Set *IPCR6* to this value by calling the function **cbus_set_state()**.

MACRO: **CBUS_MAGIC_ADDR**

PURPOSE: An offset to a location at which a VMEbus based 4000's operating system will write a defined long word value when shared memory can be written and read. The symbolic name of this value is *CBUS_DEV_READY*. This value is reset each time that the system is booted. It is not restored from a save system.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_MAGIC_ADDR
```

FUNCTION: **cbus_map_vme_addr()**

PURPOSE: Maps the indicated window of VMEbus address space as a slave space to the 4400 and configures the 4400 to read and write in the appropriate addressing mode for that window. It returns the base address of the mapped memory, in the vision processor's address space. Use this address to access the slave space.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
unsigned char *cbus_map_vme_addr(vmebase,bufferlen,space)
   unsigned char *vmebase;
   int bufferlen;
   cio_vme_master_space space
```

NOTES: This function is only available on a Cognex 4400.

vmebase: the base address of the window in VMEbus address.

bufferlen: the size of the window to access.

space: specifies the address space of the buffer at *vmebase.*

Note that, once you invoke **cbus_map_vme_addr()**, you cannot read or write over the VMEbus in any address mode except the one specified by *mode.* If you want to communicate with a window that is in a different address mode, you must first reconfigure the 4400 to communicate in that mode. To do this, use the function **csys_set_master_cfg()**

For internal reasons, the 4400 vision processor divides extended (A32) address space into *low* and *high* A32 space. These two spaces are defined as follows:

	Low A32 Space	**High A32 Space**
Ranges	0x2000,0000 - 0xFDFF,FFFF	0x0 - 0x7DFF,FFFF 0xA000,0000 - 0xFFFF,FFFF 0x8100,0000 - 0x81FF,FFFF

Under the following two conditions you must use the function **cbus_unmap_vme_addr()** to unmap all VMEbus slave windows before mapping a window in extended (A32) space:

- If you want to map high A32 address space and one or more windows that are not in high A32 space are mapped.
- If you want to map low A32 address space and a window in high A32 space is mapped.

Except for these prohibitions, you can repeatedly invoke **cbus_map_vme_addr()** to map new VMEbus windows.

If **cbus_map_vme_addr()** fails to map the requested region, it returns the value 0xFFFFFFFF.

MACRO: **CBUS_PTR_ADDR**

PURPOSE: An offset to a null-terminated array of pointers to **cbus_regs** data structures. One **cbus_regs** structure exists for each bus device that is implemented on the system. While the 4000 is polling for an image to boot, a single structure is allocated. This value is reset each time that the system is booted. It is not restored from a save system.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_PTR_ADDR
```

FUNCTION: **cbus_register_interrupter()**

PURPOSE: Registers a user action-routine on the 4000 for use over the VMEbus. This routine is intended to allow you to send interrupts to your host machine from the 4000. The action routine is called by the 4000's pseudo-serial I/O functions. It will not be called during bus-based booting or bus-based save system.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
void cbus_register_interrupter(routine_p);
   int *routine_p;
```

NOTES: Cognex supplies three functions for use in the action routine whose address you pass to the function **cbus_register_interrupter()**: **cbus_set_vme_int()**, **cbus_clear_vme_int()**, and **cbus_test_vme_int()**. You must use at least one of these functions to generate a 4000-to-host interrupt.

routine_p: points to the action routine to enable. If you specify NULL, the action routine is cancelled; you will not be able to send interrupts from the 4000 to a VMEbus host until you call this function again with a non-NULL argument.

The Cognex 4400 can request multiple interrupts simultaneously. The 4100/4200 can request only a single interrupt at a time; on the 4100/4200, you must clear an existing interrupt request, using the function **cbus_clear_vme_int()**, before setting another interrupt request active.

Your action routine will be passed a single integer argument. During pseudo-serial I/O, it is called and passed one argument. This argument will be one of the following values, defined in the file *cbus_def.h*:

- *CBUS_ADD_CHAR*, after the 4000 adds a character to the *tohost* buffer and sets *fromhost_pend* in the **cbus_regs** structure
- *CBUS_ADD_CHAR_DONE*, after the 4000 has added one or more characters to the *tohost* buffer and can no longer add characters to the *tohost* buffer and sets *fromhost_pend* in the **cbus_regs** structure
- *CBUS_RMV_CHAR*, after the 4000 removes a character from the *tovp* buffer and sets *fromvp_pend* in the **cbus_regs** structure
- *CBUS_RMV_CHAR_DONE*, after the 4000 has removed one or more characters from the *tovp* buffer and can no longer remove characters from the *tovp* buffer and sets *fromvp_pend* in the **cbus_regs** structure

STRUCTURE: **cbus_regs**

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

DEFINITION:

```
typedef struct cbus_regs{
char
   *tovp_start, *tovp_end,
   *tovp_head, *tovp_tail,
   *tohost_start, *tohost_end,
   *tohost_head, *tohost_tail;
struct
   fcb *fcb_p;
char
   *cause_int_p, cause_int
   tovp_pend, tohost_pend,
   fromvp_pend, fromhost_pend,
   vp_flags, host_flags,
   tovp_oob, tohost_oob;
   unused[3];
} cbus_regs;
```

NOTES: This data structure implements a single pseudo-serial device, */dev/busn*, where *n* is 0 or 1. It contains pointers to ring buffers in shared memory. Data output to a bus device from a 4000, using a serial output function, is written to the memory pointed to by that device's **cbus_regs** data structure. Using the same **cbus_regs** structure, the VMEbus host can then read that data. Inversely, the VMEbus host can write to the memory pointed to by a device's **cbus_regs** data structure. Then, using a serial input function, the 4000 can read the data.

Note that you can use a **cbus_regs** data structure only over the bus, from a VMEbus host. On the 4000, this structure is accessed by Cognex I/O functions; it is not accessed by application code that you write on the 4000.

During booting, one **cbus_regs** structure is available. This structure is the first entry in the array of pointers to **cbus_regs** structures pointed to by *CBUS_PTR_ADDR*. Your host can access this structure as soon as register *CBUS_IPCR6* is set to the value *CBUS_VP_POLLING*.

A second **cbus_regs** structure, pointed to by the second entry at *CBUS_PTR_ADDR*, is available after the 4000 has booted.

The **cbus_regs** structure is made up of pointers, each of which is a long word, and characters, each of which is a byte. All pointer variables in this structure are recorded as offsets from the beginning of the 4000's shared memory. To convert these addresses to virtual addresses on the host, add the host's virtual base address of the 4000's shared memory to the stored pointer. To convert such an address back to an offset, subtract this base address. (All pointer values that you store in 4000 shared memory for use by the 4000 must be of this form).

tovp_start: points to the beginning of a ring buffer of characters to send to the vision processor from the host. *tovp_start* is the address of the first character position in the buffer. The buffer can hold up to *(tohost_end - tohost_start)* bytes of data.

tovp_start is set by the 4000 when the **cbus_regs** structure is initialized. Its value remains constant.

tovp_end: points to the end of the ring buffer of characters to send to the vision processor from the host. *tovp_end* is the address of the last character position in the buffer.

tovp_end is set by the 4000 when the **cbus_regs** structure is initialized. Its value remains constant.

tovp_head: points to the next character to be read by the 4000.

tovp_head is set by the 4000 when the **cbus_regs** structure is initialized and must be modified only by the 4000's I/O protocol. It changes asynchronously.

tovp_tail: points to the next character position to be written by the host.

tovp_tail is initialized by the 4000 when the **cbus_regs** structure is initialized. It is never written by the 4000 after initialization, and must be maintained by the host. The host can update this pointer byte-by-byte, or at convenient intervals, such as after writing a block of data to the buffer.

tohost_start: points to the beginning of a ring buffer of characters to send to the host from the vision processor. *tohost_start* is the address of the first character position in the buffer. The buffer can hold up to *(tohost_end - tohost_start)* bytes of data.

tohost_start is set by the 4000 when the **cbus_regs** structure is initialized. Its value remains constant.

tohost_end: points to the end of the ring buffer of characters to send to the host from the vision processor. It is the address of the last character position in the buffer.

tohost_end is set by the 4000 when the **cbus_regs** structure is initialized. Its value remains constant.

tohost_head: points to the next character to be read by the host.

tohost_head is initialized by the 4000 when the **cbus_regs** structure is initialized. It is never written by the 4000 after initialization, and must be maintained by the VMEbus host. The host can optionally update this pointer at convenient intervals, such as after reading a block of data from the buffer.

tohost_tail: points to the next character to be written by the 4000.

tohost_tail is set by the 4000 when the **cbus_regs** structure is initialized and must be modified only by the 4000's I/O protocol. It changes asynchronously.

fcb_p: reserved for use by Cognex.

cause_int_p: points to the address on the 4000 to which the VMEbus host may write an interrupt character.

cause_int_p must be modified only by the 4000's I/O protocol.

cause_int: the value that, when written to the address pointed to by *cause_int_p*, causes a pseudo-serial interrupt on the 4000.

cause_int must be modified only by the 4000's I/O protocol. You should cause interrupts whenever you send or receive characters using bus-based pseudo-serial I/O. To cause an interrupt, first invoke the macro *CBUS_TEST_SET()* and then place the value *cause_int* at the address *cause_int_p*.

tovp_pend: indicates that characters have been added to the host-to-4000 buffer (beginning at *tovp_start*).

When it adds data to the buffer of characters to send to the 4000, the VMEbus host must set *tovp_pend* to 1 and send an interrupt to the 4000. The 4000 will set *tovp_pend* to 0 in response. Note that the VMEbus host must only set this field to 1 and the 4000 will set it only to 0.

tohost_pend: indicates that the VMEbus host has read characters from the 4000-to-host buffer.

When it is ready to receive more characters from the 4000, the VMEbus host should set *tohost_pend* to 1 and send an interrupt to the 4000. The 4000 will set *tohost_pend* to 0 in response. Note that the VMEbus host must only set this field to 1 and the 4000 will set it only to 0.

fromvp_pend: indicates that the 4000 has added characters to the 4000-to-host buffer.

The 4000 will set *fromvp_pend* to 1 when it adds characters to the buffer. The VMEbus host should reset *fromvp_pend* to 0 if it is using this field to determine when the 4000 has added characters. Note that the 4000's I/O protocol will only set this field to 1 and the VMEbus host must set it only to 0.

fromhost_pend: indicates that the 4000 has read characters from the host-to-4000 buffer (beginning at *tovp_start*).

The 4000 will set *fromhost_pend* to 1 when it reads from the data buffer. If it is using this *fromhost_pend* to determine when the 4000 has read characters, the VMEbus host should reset this field to 0 when it adds more characters to the buffer. Note that the 4000 will only set this field to 1 and the VMEbus host must set it only to 0.

vp_flags: indicates whether or not the 4000 is ready to perform an action, such as receiving data over the VMEbus. You read the value of *vp_flags* from the VMEbus host by bitwise ANDing *vp_flags* with the following constants:

- *CBUS_TOHOST_ENA:* indicates that the 4000 is ready to send data over the VMEbus. This is set by the 4000's I/O protocol when the device is opened and cleared when the device is closed.

- *CBUS_TOVP_ENA:* indicates that the 4000 is ready to receive data over the VMEbus. This is set by the 4000's I/O protocol when the device is opened and cleared when the device is closed.

- *CBUS_XFER_BOOT_IMG:* indicates that the 4000 is ready to transmit or receive a bootable image. This flag value is used for saving and restoring systems over the VMEbus. *vp_flags* is automatically set to *CBUS_XFER_BOOT_IMG* if you call the function **css_save_system()** with the argument *css_vme*. This flag is also set when the 4000 is polling the VMEbus for a boot image and the VMEbus host sets *BUS_XFER_BOOT_IMG* in the *host_flags* field.

host_flags: indicates whether or not the VMEbus host is ready to perform an action, such as receiving data over the VMEbus. You set the value of *host_flags* by ORing *host_flags* with the flag value. A single flag value is currently defined:

- *CBUS_XFER_BOOT_IMG:* enables the VMEbus host to transmit or receive a bootable image. This flag value is used for saving systems and booting over the VMEbus.

tovp_oob: can hold an *out of band signal* set by the VMEbus host. An out of band signal can be set by the host to cause a break over the bus. *tovp_oob* can have one of two values:

- *CBUS_NOSIGNAL:* indicates that no signal is pending

- *CBUS_BREAK:* indicates that the BREAK signal is pending

  When *tovp_oob* is set to *CBUS_BREAK* and an interrupt occurs, it causes the same action on the 4000 that a BREAK received over a serial line would cause. The 4000 will indicate that the signal has been received by setting *tovp_oob* to *CBUS_NOSIGNAL*.

  The VP must set *tovp_oob* only to *CBUS_NOSIGNAL*; the VMEbus host must set it only to CBUS_BREAK.

tohost_oob: reserved for future use.

unused: reserved for future use.

FUNCTION: **cbus_set_isa_int()**

PURPOSE: Generates an ISA bus interrupt request to the PC (5000 only).

REFERENCE: System Software: ISA Bus I/O

INCLUDE FILE: *cbus_isa.h*

USE:

```
int cbus_set_isa_int (level, type)
   int level, type;
```

NOTES: *level* specifies the hardware interrupt request line on the PC over which to generate the interrupt. Valid lines are 7, 9, 10, 11, 12, 15. An attempt to generate an interrupt over any other line will result in *CGEN_ERR_NOTIMP* being thrown.

type must be *cbus_edge*; any other type will result in *CGEN_ERR_NOTIMP* being thrown.

cbus_set_isa_int() sets the pending interrupt register on the 5000. The interrupt handler on the PC is responsible for clearing the pending interrupt register. The 5000 cannot generate another interrupt request until this register is cleared. Clearing this register is done by writing 0x40 to the register.

The pending interrupt register is located at an offset of 4 bytes from the configured base address of the 5000's register control block in the ISA I/O address space. The address is set by DIP switches on the 5000. The default base address of the register control block is 0x330. Therefore, the pending interrupt register is at a default address of 0x334 in the ISA I/O address space. The interrupt handler must write 0x40 to this address before the 5000 can generate another interrupt request.

Note: Only one interrupt request can be pending at any time. For example, if you generate an interrupt request over line 7, you can not generate another interrupt request over any line until the interrupt request over line 7 is cleared. Interrupt requests are cleared by the interrupt handler on the PC after servicing the request or by **cbus_clear_isa_int()**.

cbus_set_isa_int() returns 0 if the operation is successful. This function returns -1 if an interrupt is already pending. No interrupt request is generated if another request is pending.

FUNCTION: **cbus_set_state()**

PURPOSE: Sets register at offset *IPCR6* to the specified state.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:
```
void cbus_set_state(state)
   int state;
```

NOTES: *state:* a value representing the state to set. The only permissible values for this argument are *CBUS_APP_UP, CBUS_APP_DOWN*, and *CBUS_VP_RUNNING*.

This function throws the value *CGEN_ERR_BADARG* if *state* is not a permitted value.

FUNCTION: **cbus_set_vme_int()**

PURPOSE: Causes the 4000's bus controller to drive the VMEbus IRQ line active. It is supplied for use in the action routine whose address you pass to the function **cbus_register_interrupter()**.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
void cbus_set_vme_int(level,vector)
   char level,vector;
```

NOTES: *level:* must be of type **char**. It is the line (in the range 1 to 7) over which to drive the interrupt.

vector: vector must be of type *char*. If the 4000's bus controller receives an interrupt acknowledge transaction for the specified level, the byte value *vector* will be returned to the system controller and the 4000's bus controller will release the VMEbus IRQ line.

This function throws the error *CBUS_ERR_NOTIMP* on the 4100/4200 if there is an interrupt pending.

MACRO: **CBUS_SHARED_BASE_ADDRESS**

PURPOSE: The offset to the base address of all of shared memory.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:
```
#define CBUS_SHARED_BASE_ADDRESS
```

NOTES: The value of *CBUS_SHARED_BASE_ADDRESS* may change, depending on the configuration of the machine. Therefore, your code should check this value after booting a system.

MACRO: **CBUS_SHARED_LENGTH**

PURPOSE: In VMEbus I/O on a Cognex 4000, **CBUS_SHARED_LENGTH** gives the offset to the size of shared memory.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_SHARED_LENGTH
```

MACRO: **CBUS_SHARED_USER_BASE**

PURPOSE: In VMEbus I/O on a Cognex 4000, **CBUS_SHARED_USER_BASE** gives the offset to the base address of that portion of shared memory on the 4000 that is not written by Cognex software.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_SHARED_USER_BASE
```

NOTES: The value of *CBUS_SHARED_USER_BASE* may change, depending on the configuration of the machine. Therefore, your code should check this value after booting a system.

MACRO: **CBUS_SHARED_USER_LENGTH**

PURPOSE: In VMEbus I/O on a Cognex 4000, **CBUS_SHARED_USER_LENGTH** gives the offset to the size of that portion of shared memory on the 4000 that is not written by Cognex software.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_SHARED_USER_LENGTH
```

FUNCTION: **cbus_test_isa_int()**

PURPOSE: Tests for a pending ISA bus interrupt request (5000 only).

REFERENCE: System Software: ISA Bus I/O

INCLUDE FILE: *cbus_isa.h*

USE:

```
int cbus_test_isa_int();
```

NOTES: Use **cbus_test_isa_int()** to determine if an interrupt request is already pending before you issue an interrupt request. The 5000 can only generate one interrupt request to the host at a time. If you call **cbus_set_isa_int()** to generate an interrupt request when an interrupt request is already pending, **cbus_set_isa_int()** will not issue the request.

cbus_test_isa_int() returns 1 if an interrupt request is pending and 0 if no request is pending.

MACRO: **CBUS_TEST_SET()**

PURPOSE: In VMEbus I/O on a Cognex 4000, this macro waits until an interrupt can occur on the 4000 and then returns. Call it before setting a pseudo-serial interrupt.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE: `CBUS_TEST_SET(base)`

NOTES: *base* is the host's base address for the 4000 in A0 or A16 space. It should be of type: *pointer to character.*

FUNCTION: **cbus_test_vme_int()**

PURPOSE: Returns nonzero if the 4000's bus controller is requesting an interrupt on the input line number. It is supplied for use in the action routine whose address you pass to the function **cbus_register_interrupter()**.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:
```
void cbus_test_vme_int(level)
   char level;
```

NOTES: *level:* the line (in the range 1 to 7) on which to test the interrupt.

MACRO: **CBUS_TRIGGER()**

PURPOSE: Executes a trigger on the 4000 over the VMEbus. Before using this macro, you must initialize the trigger on the 4000 by calling **ctg_init()**. This macro is called on the VMEbus host.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE: `CBUS_TRIGGER(base)`

NOTES: *base* is the host's base address for the 4000 in A16 or A0 space. It should be of type *pointer to character.*

FUNCTION: **cbus_unmap_vme_addr()**

PURPOSE: Unmaps the specified window of VMEbus address space.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
int cbus_unmap_vme_addr(localbase,space)
   unsigned char *localbase;
   cio_vme_master_space space;
```

NOTES: *localbase*: the base address of the mapped memory, in the vision processor's address space. It must be a value returned by the function **cbus_map_vme_addr()**.

space: the address space of the buffer to unmap.

MACRO: **CBUS_VER_ADDR**

PURPOSE: An offset to a location that contains the version of the pseudo-serial bus communications protocol running on the 4000. This value is reset each time that the system is booted. It is not restored from a save system.

REFERENCE: System Software: VMEbus I/O

INCLUDE FILE: *cbus_def.h*

USE:

```
#define CBUS_VER_ADDR
```

VARIABLE: **ccal_default_ago**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *ccal.h*

DEFINITION:
```
ccal_video_params ccal_default_ago;
```

NOTES: *ccal_default_ago* is a specific instance of the **ccal_video_params** data structure that can be supplied to **ccal_video_setup()**.

The values for *ccal_default_ago* may not be optimum for all imaging situations, but should perform reasonably under a wide range of illumination levels.

You can use **cd_showas()** to get the values for *ccal_default_ago*. You may adjust the values, if necessary, for optimal performance, or provide your own.

FUNCTION: **ccal_scale()**

PURPOSE: Calibrate camera to subject distance.

REFERENCE: Vision Tools: Searching and Character Recognition

INCLUDE FILE: *ccal_def.h*

USE:

```
int ccal_scale(image, model, cal_params, results)
   cip_buffer *image;
   ccal_model *model;
   ccal_scale_params *cal_params;
   cse_results *results;
```

NOTES: **ccal_scale()** is used to calibrate the camera to subject distance. **ccal_scale()** repeatedly scales an input image, finds the shape score of the model in the scaled image, and returns either the scale factor that produced the highest shape score or zero if the highest shape score did not exceed the accept threshold. This information can then be used to move the camera or scale the image.

image: pointer to the image to use for calibration. The image must be at least 5-10 pixels larger than the feature it encloses.

model: pointer to either a **ccal_model** (generated by **ccal_size_init()**) or a **cse_model** to use for calibration.

cal_params: pointer to the calibration parameters.

results: if non-NULL, pointer to a **cse_results** structure.

After calibration is complete you should call the clean-up function, **ccal_size_term()**.

STRUCTURE: **ccal_scale_params**

REFERENCE: Vision Tools: Searching and Character Recognition

INCLUDE FILE: *ccal_def.h*

DEFINITION:

```
typedef struct{
   int flags;
   short accept;
   int
      low, high,
      coarse_inc, fine_inc,
      lower_bound;
   cse_params *spp;
} ccal_scale_params;
```

NOTES: **ccal_scale_params** specifies the controlling parameters of the function **ccal_scale()**.

flags: reserved for future use.

accept: the lowest shape score a feature can get and be considered found. We recommend that *accept* be in the range of 500 - 700.

low and *high:* the range of scale factors to try. *low* and *high* are specified as the scale factor times 100. If low and high are both zero, the system will set *low* to 1/4 the size of the image, and *high* to the size of the image.

coarse_inc: if nonzero, specifies the scale increment for the coarse search. If *coarse_inc* is zero the system will use a default value.

fine_inc: if nonzero, specifies the scale increment for the fine search. If *fine_inc* is zero the system will use a default value.

lower_bound: if nonzero, specifies the smallest allowed scale factor times 100. If *lower_bound* is zero, the default (1/4 the size of the image) will be used.

spp: if nonzero, points to a data structure of type *cse_params* to control the search. If *cse_params* is null, default values will be used.

FUNCTION: **ccal_size_init()**

PURPOSE: Generate a model of a character for use by size calibration.

REFERENCE: Vision Tools: Searching and Character Recognition

INCLUDE FILE: *ccal_def.h*

USE:

```
ccal_model *ccal_size_init(source)
   cse_model *source;
```

NOTES: **ccal_size_init()** returns a pointer to the generated model.

source: pointer to the **cse_model** from which the size calibration model is generated.

Since elements of the **cse_model** are shared with the generated model, the source model must not be deleted or modified after the call to **ccal_size_init()**, until **ccal_size_term()** has been called to delete the model.

The generated model must be deleted by **ccal_size_term()**. Using **ctr_delete_model()** to delete a model generated by **ccal_size_init()** may corrupt the original **cse_model**.

FUNCTION: **ccal_size_run()**

PURPOSE: Run a single pass of size calibration.

REFERENCE: Vision Tools: Searching and Character Recognition

INCLUDE FILE: *ccal_def.h*

USE:

```
int ccal_size_run(image,model)
   cip_buffer *image;
   ccal_model *model;
```

NOTES: **ccal_size_run()** assesses the size of the image and returns a value reflecting the relative size of the image. The value returned is guaranteed to fall in the range -1000 to 1000 as long as a meaningful assessment is made. The value's sign indicates the direction of deviation from nominal: if the sign is positive, the image is larger than the model and the camera must be moved farther away; if the sign is negative, the image is smaller than the model and the camera must be moved closer. If the result is10 or smaller, the system can be considered calibrated.

image: pointer to the image whose size is to be assessed.

model: pointer to the **ccal_model** used in assessing the size of the image. The **ccal_model** is provided by **ccal_size_init()** and is functionally equivalent to a **cse_model**.

If no reasonable assessment can be made, **ccal_size_run()** throws the error *CCAL_SIZE_INVALID*.

FUNCTION: **ccal_size_term()**

PURPOSE: Delete a **ccal_model** generated by **ccal_size_init()**.

REFERENCE: Vision Tools: Searching and Character Recognition

INCLUDE FILE: *ccal_def.h*

USE:
```
void ccal_size_term(model);
   cse_model *model;
```

NOTES: *model:* pointer to the **ccal_model** to be deleted.

STRUCTURE: **ccal_video_params**

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *ccal.h*

DEFINITION:

```
typedef struct{
   float lt_pct, rt_pct;
   short
      off_min, off_max,
      gain_min, gain_max,
      gain_0, off_0,
      lt_grey, rt_grey,
      tol;
   float gain_sc, off_sca, off_scb;
   int serves;
   bool trace;
}ccal_video_params;
```

NOTES: **ccal_video_params** controls the video calibration function, **ccal_video_setup()**.

lt_pct and *rt_pct:* the percent of pixels that determine the histogram's left and right computed tails, respectively.

The left tail is the grey value such that the area under the histogram (i.e. number of pixels) having smaller grey values than the left tail is a percent of the total number of pixels in the image.

The right tail is the grey value such that the area under the histogram to the right of (having larger grey values) the right tail is a percent of the total number of pixels in the image.

The tails are typically set 10 - 15 percent from the absolute ends of the range. Typical values for the 2000, or the 3000 and 4000 series in 6-bit mode, would be 10 and 54.

off_min and *off_max:* limits to which the derived offset may extend.

The legal range of values for these parameters is 0-255.

gain_min and *gain_max:* limits to which the derived gain may extend.

The legal range of values for these parameters is 0-255.

gain_0 and *off_0:* the initial settings for *caq_mode->gain* and *caq_mode->offset* for the first iteration of the servo loop.

lt_grey and *rt_grey:* the desired left and right tails of the histogram.

Gain and offset corrections will be made when

| (computed_left_grey value) - lt_grey | >tol

or

l(computed_right_grey value)- rt_grey l >tol

tol: the acceptable deviation between the computed values of the tails and the desired values (*lt_grey* and *rt_grey*).

gain_sc: the gain servo constant.

off_sca and *off_scb:* the offset servo constants.

off_sca modifies the term in the offset correction that corrects for the offset error itself. *off_scb* modifies the term in the offset correction that corrects for the effect of the interaction between gain and offset.

serves: the maximum number of servo iterations; i.e. the maximum number of times that **ccal_video_setup()** will go through the "acquire-test-adjust" loop.

trace: a boolean. When TRUE, the following state information is printed for each iteration of the servo loop:

- acquisition number;
- left and right tails;
- gain and offset;
- gain and offset errors;
- gain and offset corrections to be applied to the next acquisition.

On exit, the final values selected for gain and offset are printed.

From this state information, you can determine how to modify the control parameters in the **ccal_video_params** structure to facilitate optimal gain and offset calibration for a particular imaging situation.

FUNCTION: **ccal_video_setup()**

PURPOSE: Calibrate gain and offset for a window of the fame grabber.

REFERENCE: Image Processing: Acquisition and Display

INCLUDE FILE: *ccal.h*

GLOBALS: *ccal_default_ago*

USE:

```
bool ccal_video_setup(image,params)
   cip_buffer *image;
   ccal_video_params *params;
```

NOTES: **ccal_video_setup()** adjusts the analog gain and offset of the frame grabber to use the full range of the A/D converter without clipping.

image: a pointer to the image to be calibrated.

params: a pointer to the calibration parameters.

When the values set for gain and offset are optimal, the distribution of brightness levels occurring in a scene extends over the entire frame grabber spectrum of 64 grey levels (6 bits) (256 grey levels on the 3000/4400). The grey level assigned to the brightest part of a scene is near the high end of the range while the grey level assigned to the darkest part of the scene is near the opposite end. By spreading the visual data in a scene over a wider range of grey values, there is finer resolution in the resulting data. Subtle variations in image brightness are preserved because they are mapped to distinct digital grey values.

You can perform video calibration once at setup time or more frequently if you expect dramatic changes either in the appearance of the scene to be acquired or in the illumination. Usually the settings for the gain and offset are not critical since most Cognex vision algorithms are independent of changes in contrast.

The gain and offset values are calculated using a grey level histogram of an image. The image is acquired repeatedly until the distribution of its grey values meets user-specified criteria. The criteria are expressed in terms of the *right and left tails* of a histogram. The left tail is a grey value such that the sum of the pixels in the intensity histogram between the absolute minimum grey value (0) and this left tail value is a specified percent of the total number of pixels in the image. Likewise, the right tail is the grey value such that the sum of the pixels in the intensity histogram between this right tail value and the absolute maximum grey value is a specified percent of the total number of pixels in the image.

The desired grey level values for the left and right tails represent the *statistically significant* minimum and maximum grey values in a scene. A grey value that is lower or higher than the statistically significant minimum or maximum is likely to be an aberration. For example, a burned out sensor element in a camera or a spot on a camera lens would always generate a grey value of zero

that is not representative of any real data in a scene. **ccal_video_setup()** adjusts the gain and offset until the area of the tails in the histogram encompasses only those pixels in a scene that you consider statistically insignificant; i.e., irrelevant data.

The *desired* tails are defined by four parameters, two for each tail. These two parameters are:

- *lt_pct* or *rt_pct:* the percent of the number of pixels in the scene that make up the tail.
- *lt_grey* and *rt_grey:* the statistically significant minimum and maximum grey values.

Video calibration chooses the gain and offset settings that bring the *calculated* tails as close to the *desired* tails as possible. The operation is accomplished by a servo loop that acquires a new image, calculates the difference between the desired and the actual values, and then uses a formula to determine a correction for the gain and offset settings. The formula is somewhat complex because gain and offset are not independent variables; changing the offset also changes the effective gain.

The starting values for the global variables *caq_mode->gain* and *caq_mode->offset* are taken from *gain_0* and *off_0* in the data structure **ccal_video_params**. A grey value (intensity) histogram of the image is computed. If the computed tails of the histogram are within the tolerance, *tol*, specified in **ccal_video_params**, the function leaves *caq_mode->gain* and *caq_mode->offset* at the initial settings and returns TRUE (1). If the computed tails are not within the specified tolerances, **ccal_video_setup()** adjusts the hardware gain and offset, reacquires the image, recomputes the histogram and again checks the tails against the specified tolerances. The function continues in this loop for the number of iterations specified by the structure variable *serves*. If this limit is reached, the function will return FALSE (0), leaving *caq_mode->gain* and *caq_mode-> offset* with the setting that provided the greatest range in grey values.

A global variable, *ccal_default_ago*, with reasonable default **ccal_video_params** values is provided for use as the *params* argument to **ccal_video_setup()**. If these values prove inadequate, you can determine how to modify them by examining the state information for each iteration of the servo loop. This information is printed when you run **ccal_video_setup()** with the *trace* option selected.

ccal_video_setup() operates only on frame grabber images. The error *CGEN_ERR_BADARG* is thrown if:

- *image* is not a window of the frame grabber
- *image* == 0
- *params* == 0

A new gain value may cause the video amplifier to saturate (i.e., the computed left tail will have grey value 63, which is the maximum output of the video A/D converter). Since a gain error cannot be computed under this circumstance, either the previous gain, or about 7/8 of the previous gain will be used for the next image acquisition in this case. The offset error adjustment proceeds in any case.

FUNCTION: **cclp_apply()**

PURPOSE: Runs a caliper on an image at the point and orientation you specify.

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

USE:

```
int cclp_apply(caliper,image,x,y,angle,params,
      results)
   cclp_caliper *caliper;
   cip_buffer *image;
   int x,y;
   double angle;
   cclp_params *params;
   cclp_results *results;
```

NOTES: *caliper:* points to the caliper structure to use.

image: points to the image in which to search for features.

x and *y:* the coordinates of *image* where the caliper window will be centered (the application point).

angle: the angle, in degrees, at which to apply the caliper. Angles increase in the clockwise direction from the horizontal axis.

params: points to the run time parameters for the caliper. These include the accept threshold, number of results to find and whether diagnostic displays should be produced.

results: points to an array of results structures. The size of the array must be at least as large as the number of results requested in the **cclp_params** structure.

cclp_apply() returns zero if no features are found. Otherwise it returns the highest score found, which will also be the score of the first result in the results array.

cclp_apply() throws *CIA_ERR_DZ* if, in the **cclp_caliper** structure, you set *expected_size* to 0.0 and specify any constraint.

cclp_apply() may throw *CCLP_ERR_BADARG* if any of the arguments are invalid.

STRUCTURE: **cclp_caliper**

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

DEFINITION:

```
typedef struct {
   int search_length, proj_length;
   short ef_size,ef_leniency;
   float
      expected_size,
      cthr;
   cclp_polarity
      e1_polarity,e2_polarity;
   cclp_optimization optimize;
   short bits_per_pel;
   bool
      rotate_window,
      interpolate;
   short num_constraints;
   cclp_constraint *constraints;
   cmap_params *pixel_map;
   int
      ram_size,
      internal[2];
} cclp_caliper;
```

NOTES: **cclp_caliper** contains the parameters of an edge finding operation, such as the caliper window search length and projection length, the edge filter and scoring parameters, and edge polarity.

search_length and *proj_length:* define the dimensions of the caliper window. They should both be odd to insure a center pixel. If an even number is specified in either case, one less than that number will be used.

ef_size and *ef_leniency:* define the edge filter to run on the one-dimensional image created by the projection operation. *ef_size* defines the number of pixels to sum on each side of the *leniency* region. The sum of the pixels to the left is subtracted from the sum of the pixels to the right to determine the strength of an edge. A typical value for *ef_size* is between two and five inclusive. *ef_leniency* specifies the number of pixels to ignore at an edge. A typical value for leniency is between zero and two inclusive.

expected_size: the expected distance between edges of the target edge pair. This value is used to normalize the attribute values from the specified constraints. *expected_size* is only meaningful for edge pair evaluation.

cthr: the minimum edge contrast threshold in fractional pixel intensity. Edges with contrast below this value are ignored.

e1_polarity and *e2_polarity:* of type **cclp_polarity**, allow you to specify the required edge polarities.

```
typedef enum {
   CCLP_NEGATIVE,
   CCLP_POSITIVE,
   CCLP_INVALID,
   CCLP_DONT_CARE,
   CCLP_NO_PAIRS
} cclp_polarity;
```

CCLP_POSITIVE indicates a light to dark edge.

CCLP_NEGATIVE indicates a dark to light edge.

CCLP_DONT_CARE finds either dark to light or light to dark edges.

CCLP_NO_PAIRS for *e2_polarity* indicates that individual edges rather than edge pairs are to be found; in this case *cclp_results.e2.polarity* will be set to *CCLP_INVALID.*

optimize: of type **cclp_optimization**, specifies whether to maximize runtime speed, minimize memory use, or neither.

```
typedef enum {
   CCLP_SPACE,
   CCLP_DEFAULT,
   CCLP_SPEED
} cclp_optimization;
```

Set to *CCLP_SPEED* to optimize for runtime speed. Set to *CCLP_SPACE* to minimize memory use at the expense of runtime speed; set to *CCLP_DEFAULT* to allow the Caliper Tool to make its own selection internally. Appendix B contains timing results for all three values of *optimize* with several projection window sizes.

Note: When you set *optimize* to *CCLP_SPACE* or *CCLP_DEFAULT*, the Caliper Tool automatically calls the function **cclp_init()** each time you call **cclp_apply**. This is not the case for *CCLP_SPEED:* you must call **cclp_init()** each time you change a component of a **cclp_caliper** structure.

bits_per_pel: the number of bits per pixel: set to 6 for a 64 grey level image or 8 for a 256 grey level image.

rotate_window: a boolean that controls the type of projection. Set to TRUE to use *rotated window projection*, or FALSE to use *skewed window projection. rotate_window* is ignored for windows at 90° angles.

interpolate: a boolean: if TRUE, and the rotation angle is not divisible by 90°, the Caliper Tool will use several neighboring pixels in the caliper window to calculate the pixel for the one-dimensional image, which increases accuracy at the expense of execution time.

num_constraints: the number of elements in the constraints array.

constraints: points to an array of constraint structures. Each constraint places restrictions on the features that you want to locate.

pixel_map: if non-NULL, points to a **cmap_params** structure to use for pixel mapping (see the chapter *Pixel Mapping* in the manual *Standard System Software*). If *pixel_map* is NULL no pixel mapping will be done.

ram_size: the amount of space occupied by the caliper data structures. This field is valid immediately after a call to **cclp_init()**.

internal: reserved for internal use and must be initialized to zero.

FUNCTION: **cclp_cleanup()**

PURPOSE: Frees the internal storage associated with a **cclp_caliper** data structure.

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

USE:

```
void cclp_cleanup(caliper)
   cclp_caliper *caliper;
```

NOTES: *caliper:* points to the caliper to be cleaned up.

cclp_cleanup() may throw the following error:

- *CCLP_ERR_BADARG* if *caliper* is NULL or if a copy is made of a **cclp_caliper** structure and then an attempt is made to clean up the copy rather than the original. No error is thrown if the caliper has already been cleaned up.

STRUCTURE: **cclp_constraint**

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

DEFINITION:

```
typedef struct {
   cclp_constraint_type type;
   short y0, y1;
   short x0, x1, xc;
} cclp_constraint;
```

NOTES: **cclp_constraint** restricts the evaluation of edges by imposing constraints on the edge or edge pair. There are constraints for size, position, edge polarity, and contrast.

type: determines the type of constraint to use, and is of type **cclp_constraint**.

```
typedef enum {
   CCLP_STRADDLE,
   CCLP_POSITION,
   CCLP_ABS_POS,
   CCLP_SIZE,
   CCLP_ABS_SIZE,
   CCLP_SMALLER,
   CCLP_BIGGER,
   CCLP_CONTRAST,
   CCLP_E1_CONTRAST,
   CCLP_E2_CONTRAST
} cclp_constraint_type;
```

y0: the maximum score achieved at the normalized attribute value *x0*, and should be set to 1000.

y1: the minimum score achieved at the normalized attribute value *x1*, and must be in the range of 0 to 1000.

x0: the normalized attribute value at which the score begins to decrease. *x0* must not equal *x1*.

x1: the normalized attribute value at which the score reaches the *y1* minimum. *x0* must not equal *x1*.

xc: the normalized attribute cutoff value beyond which the score is zero. A value of zero indicates no cutoff.

STRUCTURE: **cclp_edge**

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

DEFINITION:

```
typedef struct {
   cclp_polarity polarity;
   short contrast;
   float position;
   short frac_cont;
   short reserved;
} cclp_edge;
```

NOTES: The data structure **cclp_edge** is part of **cclp_results**, and describes the feature for a results structure. **cclp_edge** holds the polarity, position, and contrast of a feature, and is filled by **cclp_apply()**.

polarity indicates the edge polarity; it can be one of the following enumeration values:

CCLP_POSITIVE indicates a dark to light edge.

CCLP_NEGATIVE indicates a light to dark edge.

CCLP_INVALID indicates the second edge, when locating single edges.

contrast indicates the difference in grey value on opposite sides of an edge. *contrast* holds the integer portion of the contrast value, and is normalized by the projection and edge filter weights so that the range is 0 to 100.

position indicates the position of the edge relative to the caliper application point, signed according to search direction.

frac_cont holds the fractional portion of the contrast value.

reserved is reserved for use by Cognex.

FUNCTION: **cclp_init()**

PURPOSE: Initialize a caliper data structure.

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

USE:

```
void cclp_init(caliper)
   cclp_caliper *caliper;
```

NOTES: *caliper:* the caliper to initialize.

cclp_init() creates or modifies the internal data structures required by a caliper. This function can be used when you allocate and initialize **cclp_caliper** and **cclp_constraints** structures or if you change any of the fields in an existing **cclp_caliper** structure (including its **cclp_constraints**).

cclp_init() may throw the following errors:

- *CCLP_ERR_BADARG* if caliper is NULL
- *CCLP_ERR_PARAMS* if any of the fields in the **cclp_caliper** or **cclp_constraint** structures are outside their valid range
- *CCLP_ERR_MEM* if the arguments specified result in an attempt to allocate more memory than is currently available

STRUCTURE: **cclp_params**

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

DEFINITION:

```
typedef struct {
   short accept, num_results;
   int flags;
} cclp_params;
```

NOTES: **cclp_params** specifies the parameters that control the function **cclp_apply()**.

accept: the accept threshold that the geometric average of the scores produced by all the constraints must exceed for the feature to be considered valid.

num_results: the number of results to find. This must be less than or equal to the size of the array of **cclp_results** structures passed to **cclp_apply()**.

flags: contains flags that enable the display of diagnostic information. To see how the diagnostics are displayed, run the demo, *caliper.v*. The following flags are defined:

CCLP_DIAG_PAUSE: Wait for a trackball button press before continuing. You can also press the return key on the keyboard instead of pressing a trackball button. Note that you must include *cif.v* to use *CCLP_DIAG_PAUSE*.

CCLP_DIAG_SHOW_WINDOW: Show the two-dimensional window used for the projection.

CCLP_DIAG_PRINT_TIMERS: Print the intermediate and total execution times.

CCLP_DIAG_SHOW_PROJ: Show the one-dimensional projected image, expanded in the y dimension.

CCLP_DIAG_PLOT_PROJ: Display a graph representing the grey levels in the projected image.

CCLP_DIAG_SHOW_FILT: Show the one-dimensional filtered image, expanded in the y dimension.

CCLP_DIAG_PLOT_FILT: Display a graph representing the filtered edges in the projected image.

CCLP_DIAG_SHOW_EDGES: Display lines indicating which edges satisfied the specified constraints.

CCLP_DIAG_PRINT_RESULTS: Print the results of the edge evaluation. Run the demo code to see the format of the printed results.

CCLP_DIAG_GRAPHICS_MASK: The flag bits corresponding to the graphics flags. All Caliper Tool graphics functions can be enabled by

using the OR operator with *flags* and this mask and disabled by using the AND operator with *flags* and the complement of this mask.

CCLP_DIAG_MASK: The flag bits corresponding to the diagnostic flags. All Caliper Tool diagnostic and display functions can be enabled by using the OR operator with *flags* and this mask and disabled by using the AND operator with *flags* and the complement of this mask.

STRUCTURE: **cclp_results**

REFERENCE: Vision Tools: Caliper Tool

INCLUDE FILE: *cclp.h*

DEFINITION:

```
typedef struct {
   bool found;
   short score;
   float position, size;
   cclp_edge e1, e2;
} cclp_results;
```

NOTES: **cclp_results** holds the results of Caliper Tool edge detection, and is filled by **cclp_apply()**. The number of results requested is specified in the *num_results* field of **cclp_params**, and is the size of the array of **cclp_results** that will be passed to **cclp_apply()**.

found: set to TRUE if the score of this edge or pair of edges exceeds the accept threshold.

score: the score of this edge or pair of edges in the range of 0 to 1000.

position: the position of the midpoint between edges as the signed distance from the caliper application point. For single edge search this is *e1*'s position.

size: contains the separation between edges in fractional pixels. For single edge search this field is zero.

e1 and *e2:* contain individual edge results. *e1*' s position is always less than *e2*'s. For single edge search only *e1* is valid (*e2*'s *polarity* field is set to *CCLP_INVALID*).

FUNCTION: **ccm_read()**

PURPOSE: Reads information from 3000/4000/5000 configuration memory (EEPROM) into a buffer.

REFERENCE: System Software: Booting

INCLUDE FILE: *ccm_defs.h*

USE:

```
void ccm_read(buffer,address,count)
   char *buffer;
   int address, count;
```

NOTES: This function is for use with the 3000, 4000, and 5000 series.

buffer: points to the buffer into which the EEPROM information is placed.

address: the starting address in EEPROM from which to read the information. *address* must be between 0 and 255.

count: the number of characters to read. *address* + *count* must be less than 256.

ccm_read() throws the error *CIO_ERR_READ* if the read fails and throws *CGEN_ERR_BADARG* if an invalid argument is supplied.

FUNCTION: **ccm_write()**

PURPOSE: Writes information from a buffer to the 3000/4000/5000 configuration memory (EEPROM).

REFERENCE: System Software: Booting

INCLUDE FILE: *ccm_defs.h*

USE:

```
void ccm_write(buffer,address,count)
   char *buffer;
   int address, count;
```

NOTES: This function is for use with the 3000, 4000, and 5000 series.

buffer: points to the buffer of information to write to EEPROM.

address: the starting address in EEPROM to which to write the information. *address* must be between 0 and 255.

count: the number of characters to write. *address* + *count* must be less than 256.

ccm_write() throws the error *CIO_ERR_WRITE* if the write fails and throws *CGEN_ERR_BADARG* if an invalid argument is supplied.

STRUCTURE: **ccol_buffer**

REFERENCE: Image Processing: Images

INCLUDE FILE: *ccol.h*

DEFINITION:

```
typedef struct{
int id;
short
   width, height,
   x_offset, y_offset,
   depth;
unsigned short flags;
cip_buffer *ch1,*ch2,*ch3,*lum;
ccol_csys csys;
void
   (*get_rgbl)(),
   (*put_rgb)();
} ccol_buffer;
```

NOTES: **ccol_buffer** defines a color buffer, which is a structure that represents a color image. A color buffer contains four channels each of which is implemented as a **cip_buffer.** The first three channels are color channels and the fourth channel is an *NTSC luminance* channel. The **ccol_buffer** structure also contains pointers to *put* and *get* functions for the color buffer, and global information about the color buffer. One color buffer can be a window of another color buffer. The **ccol_buffer** structure contains offset fields that identify the location of the origin of the window color buffer relative to the origin of the source buffer. You create a color buffer using the function **ccol_create()**, **ccol_link()**, or **ccol_window()**.

id: an identification tag that you can use for any purpose.

width: the number of columns in each **cip_buffer** channel. Each channel must have the same width.

height: the number of rows in each **cip_buffer** channel. Each channel must have the same height.

x_offset: the *x* coordinate of a window's origin relative to the origin of the source color buffer. This value is valid only if the flag *CCOL_FLAGS_X_OFFSET* is set.

y_offset: the *y* coordinate of a window's origin relative to the origin of the source color buffer. This value is valid only if the flag *CCOL_FLAGS_Y_OFFSET* is set.

depth: the pixel depth of each channel. This value is valid only if the flag *CCOL_FLAGS_DEPTH* is set.

flags: specifies the global attributes of the color buffer. The following values can be combined in *flags*:

CCOL_FLAGS_X_OFFSET: the *x_offset* field contains data that is valid for all channels.

CCOL_FLAGS_Y_OFFSET: the *y_offset* field contains data that is valid for all channels.

CCOL_FLAGS_DEPTH: the *depth* field contains data that is valid for all channels.

CCOL_FLAGS_FAST8: all channels are FAST8 grey scale images.

CCOL_FLAGS_REGULAR: all channels are regular grey scale images, that is, the distance between row addresses is constant in each image.

ch1, *ch2*, and *ch3:* point to the first, second, and third channels. Each channel's contents depend on the color system.

lum: points to the luminance channel. Unless the color system is user-defined, the contents of this channel are the NTSC luminance of the color buffer.

csys: the color system used in this buffer.

get_rgbl: a pointer to a color buffer *get* function. A get function reads a chosen pixel in the buffer, transforms it to the RGB system and returns the R, G, B, and NTSC luminance values. (For the *CCOL_CSYS_MONO* color system, the *get_rgbl()* function only returns the luminance value.) When you create a **ccol_buffer** in any color system except a user-defined system (*CCOL_CSYS_USER*), a pointer to the proper *get* function is supplied automatically.

If you are supplying your own color system, you must create a *get* function that correctly transforms the output pixel, and you must place a pointer to that function in this field. You also must use this interface to invoke any *get* function directly. The following is the required interface to this function:

```
void get_rgbl_function(col_buf, x, y, r, g, b, lum)
   ccol_buffer *col_buf;
   int x,y,*r,*g,*b,*lum;
```

put_rgb: a pointer to a color buffer *put* function. A put function takes an input pixel (in RGB format), transforms it to the correct color system, and stores it at a specified pixel location in the appropriate channels in the color buffer. When you create a **ccol_buffer** in any color system except a user-defined system (*CCOL_CSYS_USER*), a pointer to the proper *put* function is supplied automatically.

If you are supplying your own color system, you must create a *put* function that correctly transforms the input pixel, and you must place a pointer to that function in this field. You also must use this interface to invoke any *put* function directly. The following is the required interface to this function:

```
void put_rgb_function(col_buf, x, y, r, g, b)
   ccol_buffer *col_buf;
   int x, y, r, g, b;
```

FUNCTION: **ccol_copy()**

PURPOSE: Copy a source color buffer into the specified channels of a destination color buffer.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_copy(src, dst, channels)
   ccol_buffer *src, *dst;
   int channels;
```

NOTES: If the destination buffer is NULL, a color buffer is allocated and initialized with the same number of channels as the source buffer. If you supply the destination buffer, but that buffer lacks some channels that are specified as targets for copying, the channels are allocated and initialized.

src: points to the source color buffer.

dst: points to the destination color buffer.

channels: a flag word. Only the channels specified here are copied from *src* to *dst*. Select channels by ORing the following values:

CCOL_CH1_PLANE: channel 1

CCOL_CH2_PLANE: channel 2

CCOL_CH3_PLANE: channel 3

CCOL_L_PLANE: luminance channel

ccol_copy() throws the error *CCOL_ERR__CSYS_NOMATCH* if *src* and *dst* do not have the same color systems.

FUNCTION: **ccol_create()**

PURPOSE: Create and initialize a color buffer.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_create(width, height, depth,
   system, channels)
short width, height, depth;
ccol_csys system;
int channels;
```

NOTES: *width:* the width of each channel in the color buffer.

height: the height of each channel in the color buffer.

depth: the pixel depth of each channel in the color buffer.

system: the color system of the color buffer.

channels: a flag word that indicates which channels to create in the color buffer. A channel that is not selected will be initialized to NULL. Select channels by ORing the following values:

CCOL_CH1_PLANE: channel 1

CCOL_CH2_PLANE: channel 2

CCOL_CH3_PLANE: channel 3

CCOL_L_PLANE: luminance channel

If the color system is single-channel (*CCOL_CSYS_MONO*) the channel 1, 2, and 3 flags are ignored.

ccol_create() throws the following errors:

- *CGEN_ERR_BADARG* if any of the following is true:
  - *width* is larger than *CIP_MAX_WIDTH*
  - *height* is larger than *CIP_MAX_HEIGHT*
  - *depth* is larger than *CIP_MAX_DEPTH*
- *CGEN_ERR_BADPARAM*, if *system* does not specify a valid color system.

ENUMERATION: **ccol_csys**

REFERENCE: Image Processing: Images

INCLUDE FILE: *ccol.h*

DEFINITION:

```
typedef enum {
   CCOL_CSYS_RGB,
   CCOL_CSYS_CMY,
   CCOL_CSYS_XYZ,
   CCOL_CSYS_YIQ,
   CCOL_CSYS_HLS,
   CCOL_CSYS_HSV,
   CCOL_CSYS_HSI,
   CCOL_CSYS_rgb,
   CCOL_CSYS_cmy,
   CCOL_CSYS_xyz,
   CCOL_CSYS_MONO,
   CCOL_CSYS_USER
} ccol_csys;
```

NOTES: **ccol_csys** enumerates the color systems.

CCOL_CSYS_RGB: RGB space

CCOL_CSYS_CMY: cyan, magenta, yellow space, also known as *printer space*

CCOL_CSYS_XYZ: IEEE tristimulus color space

CCOL_CSYS_YIQ: NTSC color space

CCOL_CSYS_HLS: hue, luminance, saturation space, also known as *perceptual space*

CCOL_CSYS_HSV: hue, saturation, value space

CCOL_CSYS_HSI: hue, saturation, intensity space

CCOL_CSYS_rgb: red, green, blue chromaticity space

CCOL_CSYS_cmy: cyan, magenta, yellow chromaticity space

CCOL_CSYS_xyz: IEEE tristimulus chromaticity space

CCOL_CSYS_MONO: the single luminance channel

CCOL_CSYS_USER: a user-supplied color space

FUNCTION: **ccol_delete()**

PURPOSE: Delete a color buffer and all of its associated planes. **ccol_delete()** returns a pointer to the color buffer, with the planes deleted.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:
```
void ccol_delete(buf)
   ccol_buffer *buf;
```

NOTES: **ccol_delete()** only works on color buffers that were created by the functions **ccol_create()** or **ccol_link()**.

buf: points to the color buffer to delete.

FUNCTION: **ccol_delete_planes()**

PURPOSE: Delete one or more channels from a color buffer.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_delete_planes(buf,channels)
   ccol_buffer *buf;
   int channels;
```

NOTES: **ccol_delete_planes()** returns a pointer to the color buffer, from which the specified planes have been deleted.

buf: points to the color buffer containing the channels to delete.

channels: a flag word. Only the channels specified here are deleted. Select channels by ORing the following values:

CCOL_CH1_PLANE: channel 1

CCOL_CH2_PLANE: channel 2

CCOL_CH3_PLANE: channel 3

CCOL_L_PLANE: luminance channel

FUNCTION: **ccol_expand()**

PURPOSE: Takes a source color buffer and creates a destination buffer that is an expanded version of the source buffer.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_expand(src,dst,xmag,ymag,channels)
   ccol_buffer *src, *dst;
   int xmag,ymag,channels;
```

NOTES: **ccol_expand()** does not operate on the color buffer as a whole. Instead, it operates on each **cip_buffer** plane of the color buffer, using the same method as the function **cip_expand()**. Each plane in the destination color buffer is the expanded counterpart of the related plane in the source color buffer.

src: points to the source color buffer.

dst: points to the destination color buffer. If *dst* is NULL, **ccol_expand()** will allocate a color buffer with the same number of planes and with the same color system as *src*. If *dst* points to an existing color buffer, it must have the same number of planes and the same color system as *src*.

xmag: the horizontal magnification factor. It must be greater than or equal to zero.

ymag: the vertical magnification factor. It must be greater than or equal to zero.

channels: a flag word. Only the channels specified here are expanded. Select channels by ORing the following values:

- *CCOL_CH1_PLANE:* select channel 1
- *CCOL_CH2_PLANE:* select channel 2
- *CCOL_CH3_PLANE:* select channel 3
- *CCOL_L_PLANE:* select luminance channel

ccol_expand() throws the following errors:

- *CCOL_ERR_BADARG*, if in a **cip_buffer** plane pointed to from *src* or *dst* (if *dst* is supplied), any of the following is true:
  - *width* or *height* is less than 0
  - *depth* is less than 1 or greater than 32
- *CCOL_ERR_CSYS_NOMATCH*, if *dst* is supplied and either of the following is true:

- *src->csys* does not equal *dst->csys*
- *dst* does not contain a plane specified in the *channels* parameter

FUNCTION: **ccol_link()**

PURPOSE: Creates a color buffer from one to four **cip_buffer** structures. Flags in the **ccol_buffer** are set appropriately.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_link(ch1, ch2, ch3, lum, system)
   cip_buffer *ch1, *ch2, *ch3, *lum;
   ccol_csys system;
```

NOTES: *ch1:* points to the first channel.

ch2: points to the second channel.

ch3: points to the third channel.

lum: points to the luminance channel.

system: the color system.

At least one of the four channels must be non-NULL.

Note that **ccol_link()** does not check the mutual consistency of the values in each channel. For example, if the color system is *CCOL_CSYS_HSI*, *CCOL_CSYS_HLS*, or *CCOL_CSYS_HSV*, and a pixel of the saturation channel is set to zero, the corresponding hue channel pixel is undefined, and should be the constant *CCOL_PIX_UNDEFINED*.

ccol_link() always sets the *y_offset* field in the returned **ccol_buffer** structure to zero. It sets the *CCOL_FLAGS_Y_OFFSET* flag if either of the following is true:

- each component **cip_buffer** is identical
- no two component **cip_buffers** overlap

If the *x_offset* values of each component **cip_buffer** are identical, **ccol_link()** sets the *CCOL_FLAGS_X_OFFSET* flag in the returned **ccol_buffer** structure. It also sets the *x_offset* field in the results **ccol_buffer** structure to the common *x_offset* value.

ccol_link() throws the following errors:

- *CCOL_ERR_SIZE:* all non-NULL channels do not have the same width and height.
- *CGEN_ERR_BADARG:* all channels are NULL.

FUNCTION: **ccol_overlap()**

PURPOSE: Determines whether two color buffers overlap.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
BOOL ccol_overlap(buf1,buf2,xp,yp)
   ccol_buffer *buf1,*buf2;
   int *xp, *yp;
```

NOTES: Two color buffers *b1* and *b2* overlap if each **cip_buffer** channel of *b1* overlaps with its corresponding channel in *b2*. **ccol_overlap()** returns TRUE if the buffers overlap and FALSE otherwise. **ccol_overlap()** is the color analog of the function **cip_overlap()**.

buf1 and *buf2:* point to the color buffers that may overlap.

xp and *yp:* optional pointers. If these are not NULL and *buf2* overlaps *buf1*, *xp* and *yp* will point to the coordinates of the origin of *buf1* relative to *buf2*.

ccol_overlap() throws the error *CCOL_ERR_WRNG_OFFSET* if *buf1->flags* or *buf2->flags* does not have all of the following flags set: *CCOL_FLAGS_X_OFFSET*, *CCOL_FLAGS_Y_OFFSET*, and *CCOL_FLAGS_DEPTH*.

FUNCTION: **ccol_pt_xform()**

PURPOSE: Transform a single point in one color system into another color system.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
void ccol_pt_xform(x_in,y_in,z_in,lum_in,csys_in,
   x_out,y_out,z_out,lum_out,sys_out)
int x_in,y_in,z_in,lum_in;
ccol_csys csys_in;
int *x_out,*y_out,*z_out,*lum_out;
ccol_csys csys_out;
```

NOTES: *x_in, y_in, z_in,* and *lum_in:* the four values to transform.

csys_in: the color system of the input pixel.

x_out, y_out, z_out, and *lum_out:* pointers to the results. These will contain the transformed pixel.

csys_out: the color system of the output pixel.

FUNCTION: **ccol_sample()**

PURPOSE: Takes a source color buffer and creates a destination buffer of lower resolution and reduced size.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_sample(src,dst,xskip,yskip,
   xavg,yavg,channels)
ccol_buffer *src, *dst;
int xskip,yskip,xavg,yavg,channels;
```

NOTES: **ccol_sample()** uses subsampling and spatial averaging to filter the source color buffer. You can select one method or have **ccol_sample()** first subsample and then spatially average the source color buffer. **ccol_sample()** does not operate on the color buffer as a whole. Instead, it operates on each **cip_buffer** plane of the color buffer, using the same method as the function **cip_sample()**. Each plane in the destination color buffer is the sampled counterpart of the related plane in the source color buffer.

src: points to the source color buffer.

dst: points to the destination color buffer. If *dst* is NULL, **ccol_sample()** will allocate a color buffer with the same number of planes and with the same color system as *src*. If *dst* points to an existing color buffer, it must have the same number of planes and the same color system as *src*.

xskip: the horizontal dimension for the subsampling rectangle that is applied to each plane of the source buffer. This value must be nonzero. If you do not want horizontal subsampling, set this parameter to1.

yskip: the vertical dimension for the subsampling rectangle that is applied to each plane of the source buffer. This value must be nonzero. If you do not want vertical subsampling, set this parameter to1.

xavg: the horizontal dimension for the spatial averaging rectangle that is applied to each plane of the source buffer. This value must be nonzero. If you do not want horizontal spatial averaging, set this parameter to1.

yavg: the vertical dimension for the spatial averaging rectangle that is applied to each plane of the source buffer. This value must be nonzero. If you do not want vertical spatial averaging, set this parameter to1.

channels: a flag word. Only the channels specified here are subsampled or spatially averaged. Select channels by ORing the following values:

- *CCOL_CH1_PLANE:* select channel 1
- *CCOL_CH2_PLANE:* select channel 2
- *CCOL_CH3_PLANE:* select channel 3

- *CCOL_L_PLANE:* select luminance channel

Color buffers in the following color systems will produce incorrect results if spatial averaged, and should only be subsampled: *CCOL_CSYS_HLS*, *CCOL_CSYS_HSV*, *CCOL_CSYS_HSI*, *CCOL_CSYS_rgb*, *CCOL_CSYS_cmy*, and *CCOL_CSYS_xyz*. You can avoid this limitation by acquiring the color image in the *CCOL_CSYS_RGB* system, spatially averaging that image, and then converting the destination image to one of the color systems listed above, using the function **ccol_xform()**.

ccol_sample() throws the following errors:

- *CCOL_ERR_BADARG*, if in a **cip_buffer** plane pointed to from *src* or *dst*, any of the following is true:
  - *width* is less than 0
  - *height* is less than 0
  - *depth* is less than 1 or greater than 32
- *CCOL_ERR_CSYS_NOMATCH*, if *dst* is supplied and any of the following is true:
  - *src->csys* does not equal *dst->csys*
  - *dst* does not contain a plane specified in the *channels* parameter

FUNCTION: **ccol_set()**

PURPOSE: Set each specified plane of a color buffer to a specified constant value.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_set(cbuf,ch1val,ch2val,ch3val,
   lumval,channels)
ccol_buffer *cbuf;
int ch1val,ch2val,ch3val,lumval,channels;
```

NOTES: The specified values are treated as raw integers. **ccol_set()** does not transform the input values, with one exception: if a flag is set, **ccol_set()** computes the luminance channel using the contents of the other three channels.

cbuf: points to the color buffer.

ch1val, ch2val, ch3val: are the values for the three color channels. They each must be in the range 0 - 255.

lumval: is to the value for the luminance channel. It must be in the range (-1) - 255. The special value -1 instructs **ccol_set()** to calculate the luminance channel based on the contents of the other three channels, using the RGB color system.

channels is a flag word. Only the channels specified here are set. Select channels by ORing the following values:

- CCOL_CH1_PLANE: select channel 1
- CCOL_CH2_PLANE: select channel 2
- CCOL_CH3_PLANE: select channel 3
- CCOL_L_PLANE: select luminance channel

Note that **ccol_set()** does not check the validity of the values you set in the color buffer. For example, if the color system is *CCOL_CSYS_HSI*, *CCOL_CSYS_HLS*, or *CCOL_CSYS_HSV*, and the saturation channel is set to zero, the hue channel is undefined, and should be set to the constant *CCOL_PIX_UNDEFINED*.

ccol_set() throws the following errors:

- *CCOL_ERR_BADARG*, if any of the values *ch1val*, *ch2val*, or *ch3val* are not in the range 0 - 255, or if the value *lumval* is not in the range (-1) - 255.
- *CCOL_ERR_MISSING_PLANE*, if an attempt is made to set a NULL channel.

FUNCTION: **ccol_window()**

PURPOSE: Defines a color buffer that is a window of another color buffer.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_window(src,dst,w_x0,w_y0,w_width,
   w_height, channels)
      ccol_buffer*src, *dst;
      int w_x0, w_y0,w_width, w_height,channels;
```

NOTES: Color buffer windows are analogous to **cip_buffer** windows. A color buffer window is implemented as a **ccol_buffer** structure, and each **cip_buffer** in the color buffer window is a window of its counterpart **cip_buffer** in the source color buffer. You supply the origin and dimensions of the window.

You can create a window of a color buffer that has a specified subset of the parent buffer's channels. When you do this, channels that you do not select are set to NULL.

src: points to the source color buffer.

dst : points to the window of the color buffer. If *dst* is NULL, **ccol_window()** allocates a **ccol_buffer** before initializing it to point to the window.

w_x0 and *w_y0:* the coordinates of the requested window relative to the origin of the source color buffer. These coordinates may be negative: the offset of the window may be completely outside of the parent window.

w_width and *w_height:* the requested width and height of the window. They may be negative, but must be nonzero.

channels: a flag word that indicates which channels from the parent buffer to keep in the window buffer. A channel that is not selected will be initialized to NULL in the window. Select channels by ORing the following values:

- *CCOL_CH1_PLANE:* channel 1
- *CCOL_CH2_PLANE:* channel 2
- *CCOL_CH3_PLANE:* channel 3
- *CCOL_L_PLANE:* luminance channel

If the color system is single-channel (*CCOL_CSYS_MONO*) the channel 1, 2, and 3 flags are ignored.

ccol_window() uses negative *w_x0* and *w_y0* values and negative *w_height* and *w_width* values in the same way as the function **cip_window()**.

FUNCTION: **ccol_xform()**

PURPOSE: Copy and transform a source color buffer into the specified channels of another color buffer. The destination buffer may have a different color system than the source buffer.

REFERENCE: Image Processing: Basic Image Processing

INCLUDE FILE: *ccol.h*

USE:

```
ccol_buffer *ccol_xform(src, dst, dst_system, channels)
   ccol_buffer *src, *dst;
   ccol_csys dst_system;
   int channels;
```

NOTES: *src:* points to the source color buffer.

dst: points to the destination color buffer. If *dst* is NULL, a destination color buffer is allocated and initialized to the correct color system, with the specified number of color channels.

dst_system: the color system of the destination color buffer. The pixels in the source color buffer are transformed to this color system.

channels: a flag word. Only the channels in *dst* specified here will contain values transformed from *src*. Select channels by ORing the following values:

- *CCOL_CH1_PLANE:* channel 1
- *CCOL_CH2_PLANE:* channel 2
- *CCOL_CH3_PLANE:* channel 3
- *CCOL_L_PLANE:* luminance channel

Unselected channels are untouched by **ccol_xform()**.

ccol_xform() throws the following errors:

- *CCOL_ERR_NOTIMP* in the following cases:
  - *src->csys* is *CCOL_CSYS_MONO* or *CCOL_CSYS_USER*
  - *dst_system* is *CCOL_CSYS_USER*
- *CGEN_ERR_BADARG* in any of the following cases:
  - *src* is NULL
  - *height* or *width* is negative in *src->ch1*, *src->ch2*, *src->ch3*, or *src->lum*
  - *depth* is negative or greater than 32 in *src->ch1*, *src->ch2*, *src->ch3*, or *src->lum*

- *src->csys* does not equal *dst->csys*, and *ch1, ch2,* and *ch3* are not all present in both color buffers

- *CCOL_ERR_CSYS_NOINV*, if *src->csys* is *CCOL_CSYS_rgb*, *CCOL_CSYS_cmy*, or *CCOL_CSYS_xyz*, *dst->csys* is not the same as *src->csys*, and *src->lum* is NULL

- *CCOL_ERR_SIZE*, if *dst* has a different color system than *src*, and also has a different width or height than *src*

FUNCTION: **cct_catch()**

PURPOSE: Catches the signal specified as its calling argument.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct.h*

USE:
```
cct_signal cct_catch(s)
   cct_signal s;
```

NOTES: A function may call **cct_catch()** at any time to establish a handler for a **cct_signal**.

s: the **cct_signal** to catch.

cct_catch() returns zero when it is called during the normal flow of execution, such as if it is called as part of the evaluation of a conditional. **cct_catch()** returns the numeric value of the **cct_signal** caught when control is passed to it as the result of the stack being unwound.

Caution: Calls to **cct_catch()** must be embedded only in simple expressions such as: `if(s = cct_catch(error_name))`. If **cct_catch()** is called in more complex expressions, the stack may be corrupted, leading to unpredictable behavior.

Note: When **cct_catch()** returns, register values are restored to what they were at the time the **cct_catch()** was activated. For example, activating a **cct_catch()**, modifying a register variable while the catch is active, then attempting to check this variable when the **cct_catch()** returns, produces incorrect results. This is not currently an issue for V compiled code, because V currently allows but *ignores* the register storage class. There is no guarantee that future releases will ignore the register storage class.

FUNCTION: **cct_end()**

PURPOSE: Deactivates a catch function for a signal.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct.h*

USE:

```
void cct_end(s)
   cct_signal s;
```

NOTES: **cct_end()** must be called by the function that activated the **cct_catch()** being disabled, after the catch function is called. You use **cct_end()** to indicate that the calling function will no longer handle the specified error condition.

s: the **cct_signal** whose **cct_catch()** is to be deactivated.

Note: Any catch functions called between the catch function being deactivated and the call to **cct_end()** are also deactivated.

cct_end() throws the error *CCT_ERR_NOMATCH* if no matching **cct_catch()** is active in the function that calls **cct_end()**.

FUNCTION: **cct_error()**

PURPOSE: Initiates notification that an error has occurred.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct.h*

USE:

```
void cct_error(s)
   cct_signal s;
```

NOTES: **cct_error()** may be used to propagate notification that an error has occurred, but this is not recommended, because redundant stack traces may be performed.

s: the **cct_signal** to throw.

cct_error() calls the function **cct_willcatch()** to determine if a catch for a signal is active before throwing the signal. If a catch is active, **cct_error()** calls **cct_throw()** for the signal. If a catch is not active, **cct_error()** calls **cd_strace()** to print a stack trace to provide debugging information before calling **cct_throw()**.

cct_error() is intended to speed up the development and improve the maintainability of your applications by providing debugging information. Use **cct_error()** when, if a signal is not going to be caught by a catch specific to the signal, you need to know which function threw the signal, with which arguments the function was called, and which function called the function that threw the signal.

cct_error() throws the error *CGEN_ERR_BADARG* if you specify a wildcard **cct_signal** for *s*.

FUNCTION: **cct_name()**

PURPOSE: Returns the name of the **cct_signal** that corresponds to the specified constant.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct.h*

USE:
```
char *cct_name(s)
   cct_signal s;
```

NOTES: **cct_name()** searches the symbol table for a constant of the specified value whose name contains the substring *_ERR*. If it finds one, **cct_name()** returns a pointer to the name; otherwise it returns a string encoding the value in numeric form.

s: the constant whose **cct_signal** name, if any, is returned.

```
V>{printf("%s\n", cct_name(100001));}
CHP_ERR_CORRUPT

V>{printf("%s\n", cct_name(123456));}
12, 3456
```

TYPE: **cct_signal**

PURPOSE: Represents the vision processor's errors and interrupt-like events. **cct_signals** are error names you pass to the catch and throw functions.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct_sig.h*

DEFINITION: `<group>_ERR_<error>`

NOTES: **cct_signals** are positive integers with two components, a *group* code and an *error* code. The group code and error code combine to form a **cct_signal** code as follows:

```
cct_signal = (10000 * group) + error
```

where *group* and *error* are both strictly positive.

Three built in preprocessor macros simplify composition and decomposition of the group and error codes and the **cct_signal** code. These macros are:

```
/*To create a signal*/
#define CCT_SIG(g,e) (10000 * g) + (e))

/*To mask out an error code*/
#define CCT_SIG_E(s) (((int) s) / 10000)

/*To mask out a group code*/
#define CCT_SIG_G(s) (((int) s) % 10000)
```

MACRO: **CCT_SIG()**

PURPOSE: Creates a **cct_signal**.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct_sig.h*

USE:
```
#define CCT_SIG(g,e) (10000 * g) + (e))
```

NOTES: *g:* the group code.

e: the error code.

MACRO: **CCT_SIG_E()**

PURPOSE: Masks the group code out of a **cct_signal** code, leaving the error code.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct_sig.h*

USE:
```
#define CCT_SIG_E(s) (((int) s) / 10000)
```

NOTES: *s:* the **cct_signal** to mask.

MACRO: **CCT_SIG_G()**

PURPOSE: Masks the error code out of a **cct_signal** code, leaving the group code.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct_sig.h*

USE:
```
#define CCT_SIG_G(s) (((int) s) % 10000)
```

NOTES: *s:* the **cct_signal** to mask.

FUNCTION: **cct_throw()**

PURPOSE: Propagates notification that an error has occurred.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct.h*

USE:

```
void cct_throw(s)
   cct_signal s;
```

NOTES: **cct_throw()** may be used to initiate notification that an error has occurred, but this is not recommended, because no stack trace is performed if there is no catch for the error.

s: the **cct_signal** to throw.

When **cct_catch()** catches an error, notification that the error has occurred stops. You call **cct_throw()** from a function that catches an error, but does not handle the error, to notify higher level functions that the error has occurred. **cct_throw()** terminates execution of the function that calls it if the function does not contain a catch for the signal.

You should use **cct_throw()** in unwind protects. In unwind protects, you clean up a resource, terminate the function, and pass the error on to the next catch.

If you use **cct_error()** instead of **cct_throw()** in unwind protects, every unwind protect that catches the error must check to see if a catch for that error is available in a higher level function, and if none are, must do a stack trace. This could cause redundant stack traces.

cct_throw() throws the error *CGEN_ERR_BADARG* if you specify a wildcard **cct_signal** for *s*.

FUNCTION: **cct_willcatch()**

PURPOSE: Determines whether or not an active **cct_catch**() is available to catch a cct_signal.

REFERENCE: System Software: Error Handling

INCLUDE FILE: *cct.h*

USE:
```
BOOL cct_willcatch(s)
   cct_signal s;
```

NOTES: **cct_willcatch()** returns nonzero if a catch is available for *s*.

s: the **cct_signal** for which to search for an active **cct_catch()**.

cct_willcatch() ignores any **cct_catch()** call with a wildcard argument, since wildcard catches typically are used for unwind protects or other situations where the calling function does not handle a specific **cct_signal**.

TYPE: **cd_bkpt_id**

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

DEFINITION:
```
typedef int cd_bkpt_id;
```

NOTES: **cd_bkpt_id** identifies a breakpoint.

FUNCTION: **cd_bkpt()**

PURPOSE: Sets a breakpoint in a program at a specified address.

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

USE:

```
cd_bkpt_id cd_bkpt(address)
   unsigned short *address;
```

NOTES: You should save the value returned by **cd_bkpt()**. It identifies the breakpoint and must be passed to the function **cd_clear_bkpt()** to clear the breakpoint and to the function **cd_bkpt_pred()** to set a predicate function.

address: the address at which to set the breakpoint.

cd_bkpt() throws the following errors:

- *CD_ERR_TBLFULL*, if no more breakpoints are available
- *CD_ERR_BKPTSET*, if you attempt to set a breakpoint at an address that is already associated with a breakpoint
- *CD_ERR_MEMERR*, if it cannot reference the address passed to it

FUNCTION: **cd_bkpt_pred()**

PURPOSE: Associates a predicate function with a breakpoint.

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

USE:

```
c_function cd_bkpt_pred(id,pred_func_p)
   cd_bkpt_id id;
   c_function pred_func_p;
```

NOTES: The return value points to the previous predicate function associated with the breakpoint or is NULL if no predicate function was previously associated with the breakpoint.

id: the identifier of the breakpoint.

pred_func_p: points to a predicate function.

cd_bkpt_pred() throws the error *CGEN_ERR_BADARG* if *id* does not identify an breakpoint.

FUNCTION: **cd_clear_bkpt()**

PURPOSE: Clears a breakpoint.

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

USE:

```
void cd_clear_bkpt(id)
   cd_bkpt_id id;
```

NOTES: *id:* the breakpoint to clear. It must be either a **cd_bkpt_id** returned by **cd_bkpt()** or, to clear all breakpoints, the macro **CD_ALLBKPTS**.

```
#define CD_ALLBKPTS /* cd_clear_bkpt() will clear
                       all breakpoints */
```

This function throws the error *CGEN_ERR_BADARG* if *id* is not a valid parameter.

FUNCTION: **cd_clear_trace()**

PURPOSE: Clears all active function traces.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:
```
void cd_clear_trace()
```

NOTES: You should use this function only when you want to remove all function traces. You can use **cd_ftrace()** with null function and action arguments to clear the trace on a single function.

FUNCTION: **cd_dsc()**

PURPOSE: Prints a description of the specified object.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:

```
void cd_dsc(name)
   char *name;
```

NOTES: This includes the type of the object if the type has been declared, and the address of the object if the object has been defined.

name: points to the name of the object to describe. It can be any of the following:

- A global which is either a variable, a **typedef**, or a structure or union tag. Structures containing structures or structures containing indirect references to structures are not expanded.
- A function.
- A preprocessor macro. The description for the macro name is equivalent to the original definition although it may have more or less white space (blanks) than the original definition and the formal parameter names may differ.

FUNCTION: **cd_dump()**

PURPOSE: Prints a specified number of bytes in hexadecimal and ASCII.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:

```
void cd_dump(address,count)
   char *address;
   int count;
```

NOTES: *address:* points to the starting address for the dump.

count: the number of bytes to print.

For example, the following input:

```
char *string="hello";
cd_dump(string,strlen(string));
```

yields the display:

```
03659A: 68656C6C 6F hello
```

In this display, `"03659A:"` is the address in hexadecimal, `" 68656C6C 6F"` is the string value in hexadecimal, and `"hello"` is the ASCII representation of the location.

FUNCTION: **cd_ftrace()**

PURPOSE: Establishes a trace on a function.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:

```
void cd_ftrace(fn,n,action)
   unsigned short *fn;
   int n;
   int (*action) ();
```

NOTES: *fn:* points to the function to trace. At most, ten functions can be traced simultaneously.

n: the stack depth argument (**cd_strace()** is called when *fn* is called). If you specify *n* == 0, the stack is not traced.

action: an optional action routine, typically **cvc_compile()**, to execute when *fn* is called. It can perform any action needed to debug *fn*. When **cd_ftrace()** calls *action*, it supplies *action* with one argument: a pointer to the first parameter of *fn*. If you specify NULL for *action*, no action routine is called.

fn can be traced unobtrusively by setting *n* to zero (so no stack trace is performed) and providing an action routine that produces no output, but checks for an error condition or stores other information.

Specifying zero for *n* and NULL for *action* clears the trace on *fn*.

Certain deeply embedded system functions, such as **cvc_alloc()** and **free()** cannot be traced. Attempting to trace such a function will cause V to restart.

Using **cd_ftrace()** is similar to setting a breakpoint.

VARIABLE: **cd_full_debug**

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

DEFINITION:

```
int cd_full_debug;
```

NOTES: When *cd_full_debug* is set to 1, the debugger prints a display each time your program halts at a breakpoint and each time that you single step after a breakpoint. When *cd_full_debug* is 0, your program will halt and the debugger will invoke **cvc_compile()** but it will not print the display. Note that you cannot single step through assembly language code unless *cd_full_debug* is set to 1.

If the vision processor receives a *CIC_ERR_INT* signal during normal execution, and *cd_full_debug* is set to 1, the debugger will be entered as if a breakpoint were encountered.

VARIABLE: **cd_maxlevel**

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

DEFINITION:

```
int cd_maxlevel;
```

NOTES: Used to limit the recursion level of **cd_show()**. For all objects that are shown recursively, *cd_maxlevel* (default value 12) limits the recursion level. This variable prevents the display from getting too cluttered. When the recursion level reaches *cd_maxlevel* any deeper recursions are indicated only by "(more)."

STRUCTURE: **cd_pstate**

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

DEFINITION:

```
typedef struct {
   unsigned short *pc;
   short sr;
   long
      d0,d1,d2,d3,
      d4,d5,d6,d7,
      a0,a1,a2,a3,
      a4,a5,a6,sp;
   } cd_pstate;
```

NOTES: *pc*: the program counter.

sr: the status register.

d0, d1, d2, d3, d4, d5, d6, and *d7:* the data registers.

a0, a1, a2, a3, a4, a5 and *a6:* the address registers.

sp: the stack pointer.

VARIABLE: **cd_R**

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

DEFINITION:

```
cd_pstate *cd_R;
```

NOTES: *cd_R:* points to the system's **cd_pstate** data structure. *cd_R* is NULL until a breakpoint is reached or until a predicate function is called. When one of these events occurs, the system's **cd_pstate** is updated and its address is placed in *cd_R*. *cd_R* then remains active (non-NULL) until you either abort execution with the ~\ command or continue execution with the ~d command, at which point, *cd_R* becomes NULL again.

Once *cd_R* becomes non-NULL, the global **cd_pstate** structure that it points to is updated at each subsequent breakpoint and following each subsequent single step.

FUNCTION: **cd_show()**

PURPOSE: Prints the value of the specified global variable.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:

```
void cd_show(name)
   char *name;
```

NOTES: *name:* points to the variable to show.

If *name* is not a variable, but is known to *V*, then *V* prints a warning and calls **cd_dsc()** on *name*. If *name* is not known to *V*, **cd_dsc()** prints "No match" and returns.

cd_show() operates *only* on global variables where the C type information for the variable is available in the symbol table. Use **cd_showas()** to display the value of *any* address including the address of an object not defined in the symbol table.

There are functions that are known to *V*, such as the Cognex **put()** and **get()** routines, that are Cognex *internal functions*. **cd_show()** describes internal functions as "(internal function)".

FUNCTION: **cd_show_bkpt()**

PURPOSE: Lists all current breakpoints.

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

USE:
```
void cd_show_bkpt()
```

NOTES: **cd_show_bkpt()** prints each active breakpoint's **cd_bkpt_id**, the hexadecimal address associated with that breakpoint, and the nearest symbolic address associated with the breakpoint.

FUNCTION: **cd_show_pstate()**

PURPOSE: Displays the contents of a **cd_pstate** data structure.

REFERENCE: System Software: Advanced Development Environment

INCLUDE FILE: *cd_asm.h*

USE:

```
void cd_show_pstate(ps)
   cd_pstate *ps;
```

cd_show_pstate() throws the error *CGEN_ERR_BADARG* if *ps* is null.

FUNCTION: **cd_show_trace()**

PURPOSE: Prints information about all currently active function traces.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:
```
void cd_show_trace()
```

NOTES: Information displayed about the functions being traced includes: the name, the address, the stack trace argument, and the name of the action routine.

FUNCTION: **cd_showas()**

PURPOSE: Allows you to display any memory location in the format you choose.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:

```
void cd_showas(address,type)
   char *address;
   char *type;
```

NOTES: While **cd_show()** can display only objects that are defined in the symbol table, and shows them only in the format in which they were declared, **cd_showas()** is not limited in this way.

This function allows you to display the variables in a local function, as well as to display structure components that are themselves structures.

address: points to the address to show. It can be any valid memory location.

type: points to any variable or **typedef** defined in the symbol table.
cd_showas() displays *address* according to the type you specify.

FUNCTION: **cd_strace()**

PURPOSE: Prints the sequence of *n* dynamically nested function calls previous to the call to **cd_strace()**.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE:
```
void cd_strace(n)
   int n;
```

NOTES: *n:* the number of function calls to display. To display a complete stack trace, specify -1.

When a Cognex internal function is found by a call to **cd_strace()**, it is displayed as "(internal)" rather than by its name. The display includes all of the arguments to the function calls, printed in decimal notation.

FUNCTION: **cd_undefined()**

PURPOSE: Prints a display in two parts. The first part contains those symbols that are used but not defined in V code. The second part contains those symbols that are used but not defined in an *srv* file.

REFERENCE: System Software: Debugging

INCLUDE FILE: *cd.h*

USE: `void cd_undefined()`

NOTES: A misspelled variable or function name is undefined to *V*.

FUNCTION: **cdb_check_sizes()**

PURPOSE: Verifies, allocates, or reallocates RAM for an image database record structure **cdb_record**.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

USE:

```
cdb_record *cdb_check_sizes(record, data_size,
      comment_size)
   cdb_record *record;
   int data_size, comment_size;
```

NOTES: **cdb_check_sizes()** returns a volatile pointer to the checked record.

record: points to the existing record structure you want to check. To allocate a new record structure, pass in a *NULL* for *record*.

data_size: specifies the number of bytes required to hold the *data* component in a record that you want to write to the structure specified by *record*.

comment_size: specifies the number of bytes required to hold the *comment* component in a record that you want to write to the structure specified by *record*. *comment_size* should include space for the comment's *NULL* terminator.

FUNCTION: **cdb_close()**

PURPOSE: Closes an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_close(fd)
   int fd;
```

NOTES: If the IDB file was opened for any type of writing or appending, then **cdb_close()** writes a revised index and EOF record to the end of the file before closing it.

fd: is the file descriptor that identifies the file to close.

cdb_close() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file
- *CDB_ERR_SEEK*, if the software moves the file pointer beyond the beginning or end of the file

STRUCTURE: **cdb_context**

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

DEFINITION:

```
typedef struct{
   int
      x_offset, y_offset,
      width, height,
      background;
}cdb_context;
```

NOTES: **cdb_context** describes the location and size of the window in an acquired frame grabber image (*caq_image*) that was or will be saved to a database record. Only **cdb_put_acquire()**, **cdb_get_acquire()**, and **cdb_select_acquire()** use **cdb_context**.

x_offset and *y_offset:* specify the offset distance from the frame grabber origin (the top left pixel) to the origin (top left pixel) of the desired image window.

width and *height:* specify the window's width and height.

background: specifies the value assigned to pixels that are outside the desired window. When you are using **cdb_put_acquire()**, you can set this parameter to the symbolic value *CDB_CTX_AVG_BACKGR*. This value represents the average value of the pixels from the four corners of the saved window.

When you use **cdb_put_acquire()**, you can specify that the window include the entire frame grabber image or part of that image. To save the entire image, set *x_offset* and *y_offset* to zero, and set *width* and *height* to the width and height of the frame grabber.

Note that if your window extends beyond the current frame grabber size, then IDB software restricts the window to that frame grabber size.

FUNCTION: **cdb_copy_file()**

PURPOSE: Makes a copy of an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_copy_file(from_file, to_file)
   char *from_file, *to_file;
```

NOTES: **cdb_copy_file()** does *not* copy IDB records that are flagged as *deleted.*

from_file: points to the name of the source file to be copied.

to_file: points to the name of the destination file.

cdb_copy_file() throws the following errors:

- *CDB_ERR_OPEN*, if the maximum of ten files are already open.
- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly.
- *CDB_ERR_SEEK*, if the software moves the file pointer beyond the beginning or end of the file.

FUNCTION: **cdb_copy_record()**

PURPOSE: Copies an image database record structure in RAM to another IDB record structure in RAM.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

USE:

```
cdb_record *cdb_copy_record(from_record, to_record)
   cdb_record *from_record, *to_record;
```

NOTES: **cdb_copy_record()** returns a volatile pointer to the destination structure. This function uses **cdb_check_sizes()**. **cdb_copy_record()** *does* copy records that are flagged as *deleted*.

from_record: points to the record you want to copy.

to_record: points to the destination record. **cdb_check_sizes()** allocates this record if *to_record* is *NULL*.

FUNCTION: **cdb_dump_index()**

PURPOSE: Prints the index for an open image database file to the VP serial line (*stdout*).

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_dump_index(fd)
   int fd;
```

NOTES: **cdb_dump_index()** is a utility function.

fd: is the file descriptor that identifies the target file.

cdb_dump_index() throws the error *CDB_ERR_BADFD* if:

- *fd* does not identify an open IDB file
- the file does not have an index

MACRO: **CDB_E()**

PURPOSE: Returns a pointer to an index entry for a target record in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

USE: `CDB_E(f,n)`

NOTES: **CDB_E** is a low-level index access macro that usually is called by the **cdb_select_*()** functions. You also can call it directly for special index searches, when the selection functions and key functions cannot search for the target records you want to find.

f: points to the file's **cdb_file** structure. This pointer may be obtained using **cdb_get_file()**.

n: specifies the record number of the target record.

MACRO: **CDB_EK()**

PURPOSE: Returns a pointer to the user-defined *key* data in a record's index entry in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

USE:

```
CDB_EK(e)
```

NOTES: **CDB_EK** is a low-level index access macro that usually is called by the **cdb_select_*()** functions. You also can call it directly for special index searches, when the selection functions and key functions cannot search for the target records you want to find.

e: points to an index entry

STRUCTURE: **cdb_entry**

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

DEFINITION:

```
typedef struct{
   int
      record_offset, offset,
      version,
      sequence_number,
      comment_size, data_size,
      type, stype,
      data_version;
}cdb_entry;
```

NOTES: **cdb_entry** describes one entry in the index for an IDB file. Each record in an IDB file has a corresponding **cdb_entry** index entry that locates and describes the record. An optional, user-defined key structure can be appended to each **cdb_entry**, to add a record key to each index entry.

Note that the *version* through *data_version* components are identical to components in the **cdb_record** structure.

record_offset: specifies the byte offset from the start of the IDB file to the start of the record in that file.

offset: specifies the byte offset from the start of the IDB file to the start of the record's *comment* component.

version: specifies the version number for the *format* of the **cdb_record** structure. This component is read-only. The current, or most recent, version number is always *CDB_VER_NUM_CUR*.

sequence_number: specifies the record sequence number.

comment_size: specifies the actual byte size of the ASCII comments pointed to by the *comment* component. This size includes space for the comment's *NULL* terminator.

data_size: specifies the actual byte size of the data pointed to by the *data* component.

type: specifies the record type, which indicates the kind of data the record contains. Cognex reserves hex values 0 through 0x7FFFFFFF for Cognex-defined record types. You can use 0x80000000 through 0xFFFFFFFF to define new record types. A record can have only one type. The defined record types are:

CDB_IMAGE_TYPE: specifies that the record contains an image which is represented by an arbitrary **cip_buffer** data structure.

CDB_ACQUIRE_TYPE: specifies that the record contains an acquired image (*caq_image*) window and related frame grabber control and formatting data. The window may contain the entire acquired image or a subset of that image. Also see the *stype* component below for defined subtypes.

CDB_MODEL_TYPE: specifies that the record contains an image *model*, or template. Cognex applications can generate several model types, such as search models, surface mount device (SMD) models, Generalized Hough Transform (GHT) models, and so on. Use the *stype* (subtype) component below to specify the type of model.

CDB_HEADER_TYPE: specifies that the record is an IDB *file header record*, which should be the first record stored in an IDB file. The *comment* component for this record type contains an ASCII string that specifies version numbers and configuration data for the VP hardware and software, and the VP state, when the record is written. The *data* component for this record type should be *NULL*. Use **cdb_put_data()** to create this record type.

CDB_INDEX_TYPE: specifies that the record contains the index for an IDB file.

CDB_EOF_TYPE: specifies that the record contains the end-of-file (EOF) marker for an IDB file.

CDB_COMMAND_TYPE: specifies that the record contains a host command sent to the VP.

CDB_RESPONSE_TYPE: specifies that the record contains the VP response to a host command or instruction. This response is sent from the VP to the host, and is *not* a vision processing operation result (compare *CDB_RESULT_TYPE*).

CDB_RESULT_TYPE: specifies that the record contains a result or output from a vision processing operation such as searching (compare *CDB_RESPONSE_TYPE*).

CDB_PERF_TYPE: specifies that the record contains performance data. This data typically is some number of milliseconds for an action such as an inspection, acquisition, command execution, or similar event.

CDB_DELETE_TYPE: specifies that the record contents are flagged as deleted.

stype: specifies the record's *subtype*, which allows you to further classify a record type. If you want to specify a subtype, typically you must define it yourself. Cognex reserves values 0 through 0x7FFFFFFF for Cognex-defined subtypes. You can use 0x80000000 through 0xFFFFFFFF to define new subtypes.

The following subtypes exist for *CDB_ACQUIRE_TYPE* (acquired image) records only:

CDB_RUN_SUBTYPE: specifies an image acquired during a production run of your application software.

CDB_TRAIN_SUBTYPE: specifies an image acquired during model training.

CDB_CALIBRATION_SUBTYPE: specifies an image acquired to calibrate a field of view.

data_version: specifies the version number of the *packing format* used for the record's actual data.

FUNCTION: **cdb_exit()**

PURPOSE: Cleans up and closes the image database interface, including related files, RAM storage, and disk access.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_exit()
```

NOTES: After running **cdb_exit()**, you must call **cdb_init()** before resuming any IDB operations.

STRUCTURE: **cdb_file**

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

DEFINITION:

```
typedef struct{
   int flags, device, dev_fd;
   cdb_index index;
}cdb_file;
```

NOTES: **cdb_file** describes the current location and state of an IDB file and its index. You use **cdb_file** only to directly access a file's index, and then only when the **cdb_select_*()** functions and your optional **test_key()** function are unable find target records due to unusual searching criteria.

flags, *device*, and *dev_fd:* are reserved for use by Cognex, so you do not change them. *flags* describes the current state of the file and its index. *device* identifies the disk where the IDB file currently resides. *dev_fd* specifies the *int* file descriptor (fd) that identifies the file.

index: is the file's index.

FUNCTION: **cdb_free_record()**

PURPOSE: Frees RAM that was allocated for an image database record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

USE:

```
void cdb_free_record(record)
   cdb_record *record;
```

NOTES: Use **cdb_free_record()** *only* if the target record was allocated by **cdb_check_sizes()**.

record: points to the record to free.

FUNCTION: **cdb_get_acqimg()**

PURPOSE: Reads an acquired image record in an image database file, and rebuilds the image in a **cip_buffer** in the volatile IDB global record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cip_buffer *cdb_get_acqimg(fd, sub_type, comment)
   int fd;
   int *sub_type;
   char **comment;
```

NOTES: **cdb_get_acqimg()** returns a volatile pointer to the **cip_buffer** where the image is rebuilt.

The original record must have been written using **cdb_put_acquire()**. **cdb_get_acqimg()** acts like **cdb_get_image()**, rebuilding the image and ignoring the stored frame grabber data.

fd: is the file descriptor that identifies the target file.

sub_type: points to an output parameter. Upon return, it contains the subtype of the target image record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the image record. Set this argument to *NULL* to ignore the comment.

cdb_get_acqimg() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_acqimg_data()**

PURPOSE: Reads the image pixel values in a large *acquired image* record in an image database file, and rebuilds the image in an existing **cip_buffer** structure.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_get_acqimg_data(fd, record_number, image)
   int fd;
   int record_number;
   cip_buffer *image;
```

NOTES: **cdb_get_acqimg_data()** ignores any related frame grabber data stored with the image. To match the dimensions of the **cip_buffer** to the image, use **cip_create()** to create a **cip_buffer** that has the dimensions returned by **cdb_get_acqimg_hdr()** for the target record.

The **cip_buffer** must have the same depth as the retrieved image, but the width and height may vary. If the **cip_buffer** is smaller than the image, only the upper-left corner of the image is retrieved. If the **cip_buffer** is larger than the image, unused buffer pixel values are left as they were when the buffer was created or last used.

fd: is the file descriptor that identifies the target file.

record_number: specifies the record number of the target record in the IDB file. **cdb_get_acqimg_hdr()** returns this value.

image: points to an output parameter, an existing **cip_buffer**. Upon return, the buffer contains the rebuilt image.

cdb_get_acqimg_data() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.
- *CGEN_ERR_INDEX*, if the target record does not have an entry in the file's index.

FUNCTION: **cdb_get_acqimg_hdr()**

PURPOSE: Reads the record header components of a large *acquired image* record in an image database file, and copies them to a record structure in RAM.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_get_acqimg_hdr(fd, header, width, height, depth)
   int fd;
   cdb_record_hdr *header;
   int *width,*height,*depth;
```

NOTES: **cdb_get_acqimg_hdr()** is similar to **cdb_get_image_hdr()**. **cdb_get_acqimg_hdr()** starts reading from a specified file at or after the current file pointer location, when it encounters the beginning of an acquired image record header. The function returns the record number of the target record in the IDB file, or a value less than zero if the function fails.

fd: is the file descriptor that identifies the target file.

header: points to an output parameter that must be a valid record structure. Upon return, it contains the target record header components.

width: points to an output parameter. Upon return, it contains the width of the stored image.

height: points to an output parameter. Upon return, it contains the height of the stored image.

depth: points to an output parameter. Upon return, it contains the depth of the stored image.

cdb_get_acqimg_hdr() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_acquire()**

PURPOSE: Reads an acquired image record (including related frame grabber data) from an image database file, and rebuilds the image window in the frame grabber, *caq_image*.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cip_buffer *cdb_get_acquire(fd, window, sub_type, comment)
   int fd;
   cdb_context *window;
   int *sub_type;
   char **comment;
```

NOTES: **cdb_get_acquire()** returns a pointer to *caq_image*. This function retrieves a record's **cdb_context** structure to restore the image window.

fd: is the file descriptor that identifies the target file.

window: points to an output parameter. Upon return, it contains the **cdb_context** structure that defines the image window and the pixel value assigned to all pixels outside the window. If you set this argument to *NULL*, the function still rebuilds the window, but upon return, the argument does not contain the **cdb_context** structure.

sub_type: points to an output parameter. Upon return, it contains the subtype of the acquired image record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to the ASCII comment associated with the acquired image record. Set this argument to *NULL* to ignore the comment.

cdb_get_acquire() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_command()**

PURPOSE: Reads a host command record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_get_command(fd, length, sub_type, comment)
   int
      fd,
      *length,
      *sub_type;
   char **comment;
```

NOTES: **cdb_get_command()** returns a volatile pointer to the command data.

fd: is the file descriptor that identifies the target file.

length: points to an output parameter. Upon return, it contains the byte size of the host command data. Set this argument to *NULL* to ignore the length.

sub_type: points to an output parameter. Upon return, it contains the subtype of the command record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the command record. Set this argument to *NULL* to ignore the comment.

cdb_get_command() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_data()**

PURPOSE: Reads a user-defined data record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_get_data(fd, length, type, sub_type, comment,
      data_version)
   int
      fd,
      *length,
      *type, *sub_type;
   char **comment;
   int *data_version;
```

NOTES: **cdb_get_data()** returns a volatile pointer to the record's *data* component.

fd: is the file descriptor that identifies the target file.

length: points to an output parameter. Upon return, it contains the byte size of the record's *data* component. Set this argument to *NULL* to ignore the length.

type: points to an output parameter. Upon return, it contains the user-defined type of the data record.

sub_type: points to an output parameter. Upon return, it contains the subtype of the data record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the record. Set this argument to *NULL* to ignore the comment.

data_version: points to an output parameter. Upon return, it contains the version number of the *packing format* to use to unpack the record's *data* component.

cdb_get_data() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_file()**

PURPOSE: Provides direct access to an image database file structure of type **cdb_file**. **cdb_get_file()** is the only way to directly access an IDB file index.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cdb_file *cdb_get_file(fd, flag_mask)
   int fd, flag_mask;
```

NOTES: **cdb_get_file()** returns a volatile pointer to the actual **cdb_file** structure. Note that you never open files with this function, but you can use it to determine the current state of a file and its index.

fd: is the file descriptor that identifies the target file.

flag_mask: specifies the state that you want the file to be in. The following values can be bitwise OR'ed together:

CDB_FF_OPEN: means the file should be open.

CDB_FF_READ: means the file should be open for reading.

CDB_FF_WRITE: means the file should be open for writing.

CDB_FF_APPEND: means the file should be open for appending.

CDB_FF_INDEX: means the file has an index.

cdb_get_file() throws the error *CDB_ERR_BADFD* if either:

- the file is not in the state you specify with *flag_mask*
- *fd* does not identify an open IDB file

FUNCTION: **cdb_get_image()**

PURPOSE: Reads an image record from an image database file, and rebuilds the image in a **cip_buffer** in the volatile IDB global record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cip_buffer *cdb_get_image(fd, sub_type, comment)
   int fd;
   int *sub_type;
   char **comment;
```

NOTES: **cdb_get_image()** returns a volatile pointer to the **cip_buffer** that contains the rebuilt image. This function does not support image windows or related frame grabber data.

fd: is the file descriptor that identifies the target file.

sub_type: points to an output parameter. When the function returns, this parameter contains the subtype of the target image record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to the ASCII comment associated with the image record. Set this argument to *NULL* to ignore the comment.

cdb_get_image() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_image_data()**

PURPOSE: Reads the image pixel values in a large image record in an image database file, and rebuilds the image in an existing **cip_buffer** structure.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_get_image_data(fd, record_number, image)
   int fd;
   int record_number;
   cip_buffer *image;
```

NOTES: To match the dimensions of the **cip_buffer** to the image, use **cip_create()** to create a **cip_buffer** that has the dimensions returned by **cdb_get_image_hdr()** for the target record.

The **cip_buffer** must have the same depth as the retrieved image, but the width and height may vary. If the **cip_buffer** is smaller than the image, only the upper-left corner of the image is retrieved. If the **cip_buffer** is larger than the image, unused buffer pixel values are left as they were when the buffer was created or last used.

fd: is the file descriptor that identifies the target file.

record_number: specifies the record number of the target record in the IDB file. **cdb_get_image_hdr()** returns this value.

image: points to an output parameter, an existing **cip_buffer**. Upon return, the buffer contains the rebuilt image.

cdb_get_image_data() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.
- *CGEN_ERR_INDEX*, if the target record does not have an entry in the file's index.

FUNCTION: **cdb_get_image_hdr()**

PURPOSE: Reads the record header components of a large image record in an image database file, and copies them to a record structure in RAM.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_get_image_hdr(fd, header, width, height, depth)
   int fd;
   cdb_record_hdr *header;
   int *width,*height,*depth;
```

NOTES: This function is similar to **cdb_get_record_hdr()**. **cdb_get_image_hdr()** starts reading from a specified file at or after the current file pointer location, when it encounters the beginning of an image record header. The function returns the record number of the target record in the IDB file, or a value less than zero if the function fails.

fd: is the file descriptor that identifies the target file.

header: points to an output parameter that must be a valid record structure. Upon return, it contains the target record header components.

width: points to an output parameter. Upon return, it contains the width of the stored image.

height: points to an output parameter. Upon return, it contains the height of the stored image.

depth: points to an output parameter. Upon return, it contains the depth of the stored image.

cdb_get_image_hdr() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_record()**

PURPOSE: Reads a record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cdb_record *cdb_get_record(fd, record)
   int fd;
   cdb_record *record;
```

NOTES: **cdb_get_record()** returns a volatile pointer to the record structure.

fd: is the file descriptor that identifies the target file.

record: points to an output parameter. Upon return, it contains the record read from the file. You can set *record* to point to a record you allocated with **cdb_check_sizes()**. Alternately, you can set *record* to *NULL* or to *cdb_rec*, which is the pointer to the IDB global record, and the function will copy the target record to the IDB global record.

cdb_get_record() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_get_record_comment()**

PURPOSE: Reads some or all of the *comment* component of a large data, image, or acquired image record in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_get_record_comment(fd, record_number, offset,
      comment, length)
   int
      fd,
      record_number,
      offset;
   char *comment;
   int length;
```

NOTES: You can call **cdb_get_record_comment()** repeatedly to read a large ASCII comment in chunks.

fd: is the file descriptor that identifies the target file.

record_number: specifies the record number of the target record in the IDB file. **cdb_get_record_hdr()** returns this value.

offset: specifies the location to start reading in the record's comment component. Specify this offset in bytes from the start (byte 0) of the entire comment component.

comment: points to an output parameter. Upon return, it contains the comment chunk read from the file. If *comment* is *NULL*, the function returns. *comment* must point to at least (*length* +1) bytes of storage to accommodate the terminating *NULL*, or the function may overwrite RAM and cause unpredictable VP behavior. Note that when the entire comment is read, you must add the terminating *NULL*.

length: specifies the length in bytes of the current comment chunk to read.

The bytes (*offset*) to (*offset* + *length* -1) are copied to the RAM location specified by *comment*. If either (*offset*) or (*offset* + *length* -1) are outside the range of the record's entire comment component, this function throws *CGEN_ERR_BADARG*.

cdb_get_record_comment() also throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.

- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.
- *CDB_ERR_INDEX*, if the target record does not have an entry in the file's index.

FUNCTION: **cdb_get_record_data()**

PURPOSE: Reads some or all of the *data* component of a large data record in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_get_record_data(fd, record_number, offset, data,
      length)
   int
      fd,
      record_number,
      offset;
   char *data;
   int length;
```

NOTES: You can call **cdb_get_record_data()** repeatedly to read a large *data* component in chunks.

fd: is the file descriptor that identifies the target file.

record_number: specifies the record number of the target record in the IDB file. **cdb_get_record_hdr()** returns this value.

offset: specifies the location to start reading within the record's data component. Specify this offset in bytes from the start (byte 0) of the entire data component.

data: points to an output parameter. Upon return, it contains the data chunk read from the file. If *data* is *NULL*, the function returns. *data* must point to at least *length* bytes of storage, or the function may overwrite RAM and cause unpredictable VP behavior.

length: specifies the length in bytes of the current data chunk to read.

The bytes (*offset*) to (*offset* + *length* -1) are copied to the RAM location specified by *data*. If either (*offset*) or (*offset* + *length* -1) are outside the range of the record's entire data component, this function throws *CGEN_ERR_BADARG*.

cdb_get_record_data() also throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.

- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_INDEX*, if the target record does not have an entry in the file's index.

FUNCTION: **cdb_get_record_hdr()**

PURPOSE: Reads the record header components of a large data record in an image database file, and copies them to a record structure in RAM.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_get_record_hdr(fd, header)
   int fd;
   cdb_record_hdr *header;
```

NOTES: **cdb_get_record_hdr()** starts reading from a specified file at or after the current file pointer location, when it encounters the beginning of a record header. The function returns the record number of the target record in the IDB file.

fd: is the file descriptor that identifies the target file.

header: points to an output parameter that must be a valid record structure. Upon return, it contains the target record header components. Note that if this pointer is set to *NULL* when the function is called, the function may read unexpected data and cause unpredictable VP behavior.

cdb_get_record_hdr() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_get_record_hdr()** reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_get_response()**

PURPOSE: Reads a VP response record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_get_response(fd, length, sub_type, comment)
   int
      fd,
      *length,
      *sub_type;
   char **comment;
```

NOTES: **cdb_get_response()** returns a volatile pointer to the response data.

fd: is the file descriptor that identifies the target file.

length: points an output parameter. Upon return, it contains the byte size of the VP response data. Set this argument to *NULL* to ignore the length.

sub_type: points to an output parameter. Upon return, it contains the subtype of the response record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the response record. Set this argument to *NULL* to ignore the comment.

cdb_get_response() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_get_response()** reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_get_result()**

PURPOSE: Reads a VP results record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_get_result(fd, length, sub_type, comment, class)
   int
      fd,
      *length,
      *sub_type;
   char **comment;
   int *class;
```

NOTES: **cdb_get_result()** returns a volatile pointer to the results data.

fd: is the file descriptor that identifies the target file.

length: points to an output parameter. Upon return, it contains the byte size of the VP results data. Set this argument to *NULL* to ignore the length.

sub_type: points to an output parameter. Upon return, it contains the subtype of the results record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the results record. Set this argument to *NULL* to ignore the comment.

class: points to an output parameter. Upon return, it contains the grade of the VP results data, as explained for function **cdb_put_result()**. Set this argument to *NULL* to ignore the grade.

cdb_get_result() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_get_result()** reads forward beyond the end of the file or backward beyond the beginning of the file, or if a record of the appropriate type cannot be found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

VARIABLE: **cdb_global_sequence**

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

DEFINITION:

```
int cdb_global_sequence;
```

NOTES: **cdb_global_sequence** is used to assign a record sequence number to a record when it is written to an IDB file. All higher-level **cdb_put_*()** functions call **cdb_put_data()**, which assigns the value of *cdb_global_sequence* to the record, and then increments the global. A lower-level function for large records, **cdb_put_hdr()**, also assigns the value of *cdb_global_sequence* to the record and then increments the global.

cdb_init() sets **cdb_global_sequence** to 0. This value causes the next IDB record that is written to get sequence number 0.

STRUCTURE: **cdb_index**

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

DEFINITION:

```
typedef struct{
   int count, size, key_size;
   c_function make_key, test_key;
   int rd_fwd;
   char *array;
   int array_alloc;
}cdb_index;
```

NOTES: **cdb_index** describes the format of an IDB file's index. The index consists of an array of entries, where each entry is a **cdb_entry** structure optionally followed by a key structure. Each entry represents one record in the file.

count: specifies the number of entries in the index, which should be the same as the number of records in the corresponding IDB file.

size: specifies the size in bytes of each entry in the index, and includes the size specified by the *key_size* component.

key_size: specifies the size in bytes of *key* data appended to each entry in the index. If your code does not create a key structure for each entry, this component automatically gets set to 0.

make_key: points to any **make_key()** function that you write. If your code does not use a **make_key()** function, the *make_key* component automatically gets set to 0.

test_key: points to any **test_key()** function that you write. If your code does not use a **test_key()** function, the *test_key* component automatically gets set to 0.

rd_fwd: ("read forward") specifies the record number of the record that would be read by a call to **cdb_get_record()**.

array: points to the array of index entries.

array_alloc: specifies the number of bytes allocated for the array. If *array_alloc* is too small for an index being constructed, then IDB code automatically creates a larger array, copies the smaller array to the larger array, and deletes the smaller array.

FUNCTION: **cdb_init()**

PURPOSE: Initializes the IDB interface.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE: `void cdb_init()`

NOTES: **cdb_init()** initializes the IDB interface. This includes setting global variable *cdb_global_sequence* to 0, and allocating the global record that is pointed to by *cdb_rec*. You *must* call **cdb_init()** before calling any other **cdb_*()** function.

FUNCTION: **cdb_key_open()**

PURPOSE: Opens an image database file and supports optional IDB key functions.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_key_open(filename, mode, key_size, make_key,
      test_key)
   char *filename, *mode;
   int key_size;
   c_function make_key, test_key;
```

NOTES: **cdb_key_open()** extends the capabilities of **cdb_open()** by supporting the optional functions **make_key()** and **test_key()** that you can write. **make_key()** labels each record's index entry with a key, so you can use a **cdb_select_*()** function and **test_key()** to search and retrieve records based on their keys.

cdb_key_open() opens an image database file and returns a file descriptor of type *int*. A return value less than zero indicates an error.

Note that the *filename* and *mode* arguments are the same as for **cdb_open()**.

filename: is the MS-DOS compatible IDB file name. Specify the file name as the drive letter and colon, followed by the file path name.

mode: is the mode in which to open the file, and may be *one* of the following. Note that these modes generally follow the modes for the standard **fopen()** C function, with minor variations. *Seeking* means reading to search for particular records. The modes are:

"r" Open the file only for *reading* and seeking. If the file does not exist, return a value of -1.

"w" Create and open the file for *writing* only. If the file already exists, overwrite it.

"a" Open the file for *appending* only. Precede all write operations with a seek to the end-of-file. If the file does not exist, create it.

"r+" Open the file for reading, seeking, and *selective* rewriting. If the file does not exist, create it. If the file does exist, then you can overwrite an existing record in the file with a new record of *exactly* the same size.

"w+" Open the file for reading, seeking, and *complete* rewriting. If the file does not exist, create it. If the file does exist, then writing to it overwrites the *entire* file.

"a+" Open the file for reading, seeking, and *appending*. Precede all write operations with a seek to the end-of-file. If the file does not exist, create it.

key_size: specifies the number of bytes of *key* data that are required.

make_key: specifies the address of an optional function that you can write. This function creates an identifying *key* for a record.

test_key: specifies the address of an optional function that you can write. This function compares two keys and specifies whether or not they are the same.

Note that the next two lines of code are functionally the same:

```
cdb_open(filename, mode);

cdb_key_open(filename, mode, 0, 0, 0);
```

cdb_key_open() throws the following errors:

- *CDB_ERR_OPEN*, if the maximum of ten files are already open, or if the function returns a value less than zero.
- *CDB_ERR_WRITE*, if there is a low-level software failure while opening a file for writing or appending.
- *CDB_ERR_READ*, if there is a low-level software failure while opening a file for reading.
- CDB_ERR_MAGIC, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_MODE*, if you specify an invalid value for the *mode* argument.
- *CDB_ERR_SEEK*, if the software moves the file pointer beyond the beginning or end of the file.

FUNCTION: **cdb_open()**

PURPOSE: Opens an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_open(filename, mode)
   char *filename, *mode;
```

NOTES: **cdb_open()** opens an IDB file and returns a file descriptor of type *int* that identifies the file. A return value less than zero indicates an error.

filename: is the MS-DOS compatible IDB file name. Specify the file name as the drive letter and colon, followed by the file path name.

mode: is the mode in which to open the file, and may be *one* of the following. Note that these modes generally follow the modes for the standard **fopen()** C function, with minor variations. *Seeking* means reading to search for particular records. The modes are:

"r" Open the file only for *reading* and seeking. If the file does not exist, return a value of -1.

"w" Create and open the file for *writing* only. If the file already exists, overwrite it.

"a" Open the file for *appending* only. Precede all write operations with a seek to the end-of-file. If the file does not exist, create it.

"r+" Open the file for reading, seeking, and *selective* rewriting. If the file does not exist, create it. If the file does exist, then you can overwrite an existing record in the file with a new record of *exactly* the same size.

"w+" Open the file for reading, seeking, and *complete* rewriting. If the file does not exist, create it. If the file does exist, then writing to it overwrites the *entire* file.

"a+" Open the file for reading, seeking, and *appending*. Precede all write operations with a seek to the end-of-file. If the file does not exist, create it.

Note that the next two lines of code are functionally the same:

```
cdb_open(filename, mode);

cdb_key_open(filename, mode, 0, 0, 0);
```

cdb_open() throws the following errors:

- *CDB_ERR_OPEN*, if the maximum of ten files are already open, or if the function returns a value less than zero.

- *CDB_ERR_WRITE*, if there is a low-level software failure while opening a file for writing or appending.
- *CDB_ERR_READ*, if there is a low-level software failure while opening a file for reading.
- CDB_ERR_MAGIC, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_MODE*, if you specify an invalid value for the *mode* argument.
- *CDB_ERR_SEEK*, if the software moves the file pointer beyond the beginning or end of the file.

FUNCTION: **cdb_put_acquire()**

PURPOSE: Writes an acquired image window from the frame grabber (*caq_image*) to a record in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_acquire(fd, window, sub_type, comment)
   int fd;
   cdb_context *window;
   int sub_type;
   char *comment;
```

NOTES: **cdb_put_acquire()** uses a **cdb_context** structure to specify a *caq_image* window. This function also stores related frame grabber control and formatting data from the global acquisition variables *caq_mode* and *caq_format*.

fd: is the file descriptor that identifies the target file.

window: points to the **cdb_context** structure that defines the *caq_image* window to save, and that specifies the pixel value to assign to all pixels outside the window. To save the entire image in a window, see structure **cdb_context**.

sub_type: specifies a subtype for the acquired image record to be written.

comment: points to an ASCII comment that you can write to the acquired image record. Set this argument to *NULL* to omit the comment.

cdb_put_acquire() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_command()**

PURPOSE: Writes a host command record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_command(fd, command, length, sub_type,
      comment)
   int fd;
   char *command;
   int length;
   int sub_type;
   char *comment;
```

NOTES: Host commands are commands sent from the host to the VP.

fd: is the file descriptor that identifies the target file.

command: points to the host command data.

length: specifies the byte size of the host command data.

sub_type: specifies a subtype for the command record to be written.

comment: points to an ASCII comment that you can write to the command record. Set this argument to *NULL* to omit the comment.

cdb_put_command() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_data()**

PURPOSE: Writes a user-defined data record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_data(fd, data, length, type, sub_type,
      comment, data_version)
   int fd;
   char *data;
   int
      length,
      type,
      sub_type;
   char *comment;
   int data_version;
```

NOTES: Use **cdb_put_data()** when data doesn't fit the available IDB record type formats, but *not* when data records won't fit in available memory.

All *higher-level* **cdb_put_*()** functions call **cdb_put_data()**, which sets the record's sequence number to the value of global *cdb_global_sequence*, and then increments the global. **cdb_put_data()** then calls **cdb_put_record()**, which writes the record.

fd: is the file descriptor that identifies the target file.

data: points to the data to write to the record.

length: specifies the byte size of the data to be written.

type: specifies your own record type to be written.

sub_type: specifies the subtype for the record to be written.

comment: points to an ASCII comment that you can write to the record. Set this argument to *NULL* to omit the comment.

data_version: specifies the version number of the *packing format* used on the data pointed to by the *data* argument.

cdb_put_data() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_hdr()**

PURPOSE: Calls **cdb_put_record_hdr()** and passes to it the record header components that you want to write to a large data record in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_hdr(fd, data_size, type, subtype,
      comment_size, data_version)
   int fd, data_size, type, subtype, comment_size,
      data_version;
```

NOTES: **cdb_put_hdr()** sets the record's sequence number to the value of *cdb_global_sequence*, and then increments the global.

fd: is the file descriptor that identifies the target file.

data_size: specifies the byte size of the *complete* data component, which will be written to the record as a whole or in chunks using **cdb_put_record_data()**.

type: specifies the record type for the record to be written.

sub_type: specifies the subtype for the record to be written.

comment_size: specifies the byte size of the *complete* ASCII comment component, which will be written to the record as a whole or in chunks using **cdb_put_record_comment()**. Note that this size does *not* have to include space for the comment's *NULL* terminator. Set this argument to 0 if the record does not have a comment.

data_version: specifies the version number of the packing format used for the data component to be written.

cdb_put_hdr() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_image()**

PURPOSE: Writes an image to a record in an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_image(fd, image, sub_type, comment)
   int fd;
   cip_buffer *image;
   int sub_type;
   char *comment;
```

NOTES: **cdb_put_image()** does not support image windows or related frame grabber data.

fd: is the file descriptor that identifies the target file.

image: points to the **cip_buffer** structure that holds the image data.

sub_type: specifies a subtype for the image record to be written.

comment: points to an ASCII comment that you can write to the image record. Set this argument to *NULL* to omit the comment.

cdb_put_image() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_record()**

PURPOSE: Writes a record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_put_record(fd, record)
   int fd;
   cdb_record *record;
```

NOTES: **cdb_put_record()** fills in the record's *version* field. For details on this field, see structure **cdb_record**.

fd: is the file descriptor that identifies the target file.

record: points to the record to write to the file.

cdb_put_record() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_record_comment()**

PURPOSE: Writes the *comment* component of a large data record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_put_record_comment(fd, comment, length)
   int fd;
   char *comment;
   int length;
```

NOTES: You can call **cdb_put_record_comment()** repeatedly to write a large comment in chunks.

fd: is the file descriptor that identifies the target file.

comment: points to the current location in the ASCII comment. To omit the comment, you can either set this argument to *NULL*, or not call this function.

length: specifies the length in bytes of the current comment chunk to write.

cdb_put_record_comment() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.
- *CGEN_ERR_BADARG*, if the total comment bytes written during one run of this function, or during a complete set of runs of this function, are greater than the *comment_size* argument to **cdb_put_hdr()** for the current record.

If *fewer* than *comment_size* total bytes are written during a complete set of runs of **cdb_put_record_comment()**, then the file remains locked to access by other functions.

FUNCTION: **cdb_put_record_data()**

PURPOSE: Writes the *data* component of a large data record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_put_record_data(fd, data, length)
   int fd;
   char *data;
   int length;
```

NOTES: You can call **cdb_put_record_data()** repeatedly to write a large *data* component in chunks.

fd: is the file descriptor that identifies the target file.

data: points to the current location in the data.

length: specifies the length in bytes of the current data chunk to write.

cdb_put_record_data() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.
- *CDB_ERR_FILE_LOCKED*, if the total *comment* bytes written for this record are fewer than the *comment_size* argument to **cdb_put_hdr()**.
- *CGEN_ERR_BADARG*, if the total *data* bytes written during one run of this function, or during a complete set of runs of this function, are greater than the *data_size* argument to **cdb_put_hdr()**.

If fewer than *data_size* total bytes are written during a complete set of runs of this function, then the file remains locked to access by other functions.

FUNCTION: **cdb_put_record_hdr()**

PURPOSE: Writes a record's header components to a large data record in an IDB file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
void cdb_put_record_hdr(fd, header)
   int fd;
   cdb_record_hdr *header;
```

NOTES: This function locks all other access to the target file until the *complete* comment and data components are written, as a whole or in chunks, using **cdb_put_record_comment()** and **cdb_put_record_data()**. **cdb_put_record_data()** must be called to unlock the file.

fd: is the file descriptor that identifies the target file.

header: points to the record header containing the component values to write to the file.

cdb_put_record_hdr() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_response()**

PURPOSE: Writes a VP response record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_response(fd, response, length, sub_type,
      comment)
   int fd;
   char *response;
   int length;
   int sub_type;
   char *comment;
```

NOTES: VP responses are responses sent from the VP to the host.

fd: is the file descriptor that identifies the target file.

response: points to the VP response data.

length: specifies the byte size of the VP response data.

sub_type: specifies a subtype for the response record to be written.

comment: points to an ASCII comment that you can write to the response record. Set this argument to *NULL* to omit the comment.

cdb_put_response() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

FUNCTION: **cdb_put_result()**

PURPOSE: Writes a VP results record to an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

GLOBALS: *cdb_global_sequence*

USE:

```
void cdb_put_result(fd, result, length, sub_type, comment,
      class)
   int fd;
   char *result;
   int length;
   int sub_type;
   char *comment;
   int class;
```

NOTES: VP results are output from a vision processing operation, such as searching results. You also can use **cdb_put_result()** to write a *grade*, or an evaluation of a result, to an existing results record in an IDB file.

fd: is the file descriptor that identifies the target file.

result: points to the VP results data.

length: specifies the byte size of the VP results data.

sub_type: specifies a subtype for the results record to be written.

comment: points to an ASCII comment that you can write to the results record. Set this argument to *NULL* to omit the comment.

class: is used to grade the VP results data. You specify class by bitwise OR'-ing whichever of the following values apply:

CDB_POSITIVE: means the results indicate a successful vision processing operation. If this value is not asserted, the operation failed (negative result).

CDB_KNOWN: means the results have been examined by a person. If this value is not asserted, the results have not been examined, and the values of *CDB_TRUE*, *CDB_EXTENDED*, and *CDB_RIGHT* are not valid for the record.

CDB_TRUE: means that a person has decided that the positive or negative determination is correct. If *CDB_TRUE* is not asserted, then the positive or negative determination is wrong.

CDB_EXTENDED: means extended grading is supported for the record. If this value is not asserted, then the value of *CDB_RIGHT* is not valid for the record.

CDB_RIGHT: means additional details of the result are correct, independent of the value of *CDB_POSITIVE*. If *CDB_RIGHT* is not asserted, additional details of the result are incorrect.

cdb_put_result() throws the following errors:

- *CDB_ERR_WRITE*, if there is a low-level software failure while writing.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.
- *CDB_ERR_REWRITE*, if you are overwriting an existing record with a new record that has a different size.

STRUCTURE: **cdb_record**

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h*

DEFINITION:

```
typedef struct{
   int
      version,
      sequence_number,
      comment_size, data_size,
      type, stype,
      data_version;
   char *comment, *data;
   int comment_alloc, data_alloc;
}cdb_record;
```

NOTES: **cdb_record** describes one record in an IDB file. The *version* through *data_version* components are the record header, and also appear the record's entry (**cdb_entry**) in the IDB file index.

version: specifies the version number for the *format* of the **cdb_record** structure. This component is read-only. The current, or most recent, version number is always *CDB_VER_NUM_CUR*.

sequence_number: specifies the record sequence number.

comment_size: specifies the actual byte size of the ASCII comments pointed to by the *comment* component. This size includes space for the comment's *NULL* terminator.

data_size: specifies the actual byte size of the data pointed to by the *data* component.

type: specifies the record type, which indicates the kind of data the record contains. Cognex reserves hex values 0 through 0x7FFFFFFF for Cognex-defined record types. You can use 0x80000000 through 0xFFFFFFFF to define new record types. A record can have only one type. The defined record types are:

CDB_IMAGE_TYPE: specifies that the record contains an image which is represented by an arbitrary **cip_buffer** data structure.

CDB_ACQUIRE_TYPE: specifies that the record contains an acquired image (*caq_image*) window and related frame grabber control and formatting data. The window may contain the entire acquired image or a subset of that image. Also see the *stype* component below for defined subtypes.

CDB_MODEL_TYPE: specifies that the record contains an image *model*, or template. Cognex applications can generate several model types, such as search models, surface mount device (SMD) models, Gener-

alized Hough Transform (GHT) models, and so on. Use the *stype* (subtype) component below to specify the type of model.

CDB_HEADER_TYPE: specifies that the record is an IDB *file header record*, which should be the first record stored in an IDB file. The *comment* component for this record type contains an ASCII string that specifies version numbers and configuration data for the VP hardware and software, and the VP state, when the record is written. The *data* component for this record type should be *NULL*. Use **cdb_put_data()** to create this record type.

CDB_INDEX_TYPE: specifies that the record contains the index for an IDB file.

CDB_EOF_TYPE: specifies that the record contains the end-of-file (EOF) marker for an IDB file.

CDB_COMMAND_TYPE: specifies that the record contains a host command sent to the VP.

CDB_RESPONSE_TYPE: specifies that the record contains the VP response to a host command or instruction. This response is sent from the VP to the host, and is *not* a vision processing operation result (compare *CDB_RESULT_TYPE*).

CDB_RESULT_TYPE: specifies that the record contains a result or output from a vision processing operation such as searching (compare *CDB_RESPONSE_TYPE*).

CDB_PERF_TYPE: specifies that the record contains performance data. This data typically is some number of milliseconds for an action such as an inspection, acquisition, command execution, or similar event.

CDB_DELETE_TYPE: specifies that the record contents are flagged as deleted.

stype: specifies the record's *subtype*, which allows you to further classify a record type. If you want to specify a subtype, typically you must define it yourself. Cognex reserves values 0 through 0x7FFFFFFF for Cognex-defined subtypes. You can use 0x80000000 through 0xFFFFFFFF to define new subtypes.

The following subtypes exist for *CDB_ACQUIRE_TYPE* (acquired image) records only:

CDB_RUN_SUBTYPE: specifies an image acquired during a production run of your application software.

CDB_TRAIN_SUBTYPE: specifies an image acquired during model training.

CDB_CALIBRATION_SUBTYPE: specifies an image acquired to calibrate a field of view.

data_version: specifies the version number of the *packing format* used for the data pointed to by the *data* component.

comment: points to a *NULL*-terminated, ASCII text comment for the record.

data: points to the actual data for the record. This data may include an image.

comment_alloc: specifies the number of bytes allocated in VP RAM for the *comment* component. *comment_alloc* is 0 if the *comment* component is *NULL*.

data_alloc: specifies the number of bytes allocated in VP RAM for the *data* component. *data_alloc* is 0 if the *data* component is *NULL*.

FUNCTION: **cdb_seek()**

PURPOSE: Locates a specified image database record by moving a file pointer to a particular location in the file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_seek(fd, offset, origin)
   int fd, offset, origin;
```

NOTES: **cdb_seek()** returns the resulting file pointer offset, which always is specified from the *beginning* of the file in *bytes.* If an error occurs, the function instead returns a negative number.

fd: is the file descriptor that identifies the target file.

offset: is the offset distance you want to move the file pointer, measured from a reference location in the target file. The *origin* parameter specifies the reference location and the type of units for *offset. offset* can be specified in two different units. By default *offset* is the *byte* offset from the *origin.* Alternately, *offset* can be the *record* offset (the number of records) from the *origin,* if the symbolic value *CDB_SO_REC* is asserted in *origin.* Negative values for *offset* move the file pointer backwards in the file, and positive values move the pointer forwards.

origin: is a symbolic value that represents a reference location in the target file. Values can be one of:

CDB_SO_ABS: (*absolute* location) specifies the start of the file

CDB_SO_REL: (*relative* location) specifies the location to which the file pointer currently points

CDB_SO_EOF: (*EOF* location) specifies the end of the file

In addition, the value *CDB_SO_REC* can be bitwise OR'ed with any of the three values just listed. This value indicates that *offset* is specified in records, rather than in bytes. *CDB_SO_REC* requires that a working index must exist for the target file.

cdb_seek() throws the following errors:

- *CDB_ERR_BADFD,* if *fd* does not identify an open IDB file.
- *CDB_ERR_INDEX,* if *CDB_SO_REC* is asserted in *origin* and the file doesn't have an index. Use **cdb_copy_file()**, which copies the file and generates an index for the new file.
- *CDB_ERR_SEEK,* if the software moves the file pointer beyond the beginning or end of the file.

FUNCTION: **cdb_seek_record()**

PURPOSE: Scans forward or backward in an image database file index to find the first record that matches a given template record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
int cdb_seek_record(fd, template, mode)
   int fd;
   cdb_record *template;
   int mode;
```

NOTES: If **cdb_seek_record()** finds a matching record, it returns the record number of that record, and moves the file pointer so it points to the start of that record. If no matching record is found, the function returns a value less than zero, and does not move the file pointer.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index.

You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

cdb_seek_record() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_seek_record()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_acqimg()**

PURPOSE: Selects and reads an acquired image record in an image database file, and rebuilds the image in a **cip_buffer** in the volatile global IDB record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cip_buffer *cdb_select_acqimg(fd, template, mode,
      sub_type, comment)
   int fd;
   cdb_record *template;
   int mode;
   int *sub_type;
   char **comment;
```

NOTES: **cdb_select_acqimg()** returns a volatile pointer to the rebuilt image. The original record must have been written using **cdb_put_acquire()**. **cdb_select_acqimg()** acts like **cdb_select_image()**, rebuilding the image and ignoring the stored frame grabber data.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

sub_type: points to an output parameter. Upon return, it contains the subtype of the selected record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to the ASCII comment associated with the selected record. Set this argument to *NULL* to ignore the comment.

cdb_select_acqimg() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_acqimg()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_acquire()**

PURPOSE: Selects and reads an acquired image record (including related frame grabber data) from an image database file, and rebuilds the image window in the frame grabber, *caq_image*.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cip_buffer *cdb_select_acquire(fd, template, mode, window,
      sub_type, comment)
   int fd;
   cdb_record *template;
   int mode;
   cdb_context *window;
   int *sub_type;
   char **comment;
```

NOTES: **cdb_select _acquire()** returns a pointer to *caq_image*. This function retrieves a record's **cdb_context** structure to restore the image window.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

window: points to an output parameter. Upon return, it contains the **cdb_context** structure that defines the image window and the pixel value to assign to all pixels outside the window. If you set this argument to *NULL*, the function still rebuilds the window, but upon return, the argument does not contain the **cdb_context** structure.

sub_type: points to an output parameter. Upon return, it contains the subtype of the selected acquired image record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the selected acquired image record. Set this argument to *NULL* to ignore the comment.

cdb_select_acquire() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_acquire()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_command()**

PURPOSE: Selects and reads a host command record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_select_command(fd, template, mode, length,
      sub_type, comment)
   int fd;
   cdb_record *template;
   int
      mode,
      *length,
      *sub_type;
   char **comment;
```

NOTES: **cdb_select_command()** returns a volatile pointer to the command data.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

length: points to an output parameter. Upon return, it contains the byte size of the host command data. Set this argument to *NULL* to ignore the length.

sub_type: points to an output parameter. Upon return, it contains the subtype of the selected command record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to the ASCII comment associated with the selected command record. Set this argument to *NULL* to ignore the comment.

cdb_select_command() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_command()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_data()**

PURPOSE: Selects and reads a user-defined data record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_select_data(fd, template, mode, length, type,
      sub_type, comment, data_version)
   int fd;
   cdb_record *template;
   int
      mode,
      *length,
      *type,
      *sub_type;
   char **comment;
   int *data_version;
```

NOTES: **cdb_select_data()** returns a volatile pointer to the record's *data* component.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

length: points to an output parameter. Upon return, it contains the byte size of the data in the record's *data* component. Set this argument to *NULL* to ignore the length.

type: points to an output parameter. Upon return, it contains the user-defined record type.

sub_type: points to an output parameter. Upon return, it contains the subtype of the data record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the data record. Set this argument to *NULL* to ignore the comment.

data_version: points to an output parameter. Upon return, it contains the version number of the current *packing format* to use to unpack the record's *data* component.

cdb_select_data() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if the function reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if the function's *fd* argument does not identify an open IDB file.

FUNCTION: **cdb_select_image()**

PURPOSE: Selects and reads an image record from an image database file, and rebuilds the image in a **cip_buffer** in the volatile global IDB record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cip_buffer *cdb_select_image(fd, template, mode, sub_type,
      comment)
   int fd;
   cdb_record *template;
   int mode;
   int *sub_type;
   char **comment;
```

NOTES: **cdb_select_image()** returns a volatile pointer to the rebuilt image. This function does not support image windows or related frame grabber data.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

sub_type: points to an output parameter. When the function returns, this parameter contains the subtype of the selected image record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to the ASCII comment associated with the selected image record. Set this argument to *NULL* to ignore the comment.

cdb_select_image() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_image()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_record()**

PURPOSE: Scans forward or backward in an image database file index to find the first record that matches a given template record.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
cdb_record *cdb_select_record(fd, template, mode)
   int fd;
   cdb_record *template;
   int mode;
```

NOTES: **cdb_select_record()** copies the target record to the volatile IDB global record, and then returns a volatile pointer to that global record. All the other **cdb_select_*()** (IDB selection) functions call **cdb_select_record()** to find target records.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index.

You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

cdb_select_record() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_record()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_response()**

PURPOSE: Selects and reads a VP response record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_select_response(fd, template, mode, length,
      sub_type, comment)
   int fd;
   cdb_record *template;
   int
      mode,
      *length,
      *sub_type;
   char **comment;
```

NOTES: **cdb_select_response()** returns a volatile pointer to the response data.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

length: points to an output parameter. Upon return, it contains the byte size of the VP response data. Set this argument to *NULL* to ignore the length.

sub_type: points to an output parameter. Upon return, it contains the subtype of the response record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the response record. Set this argument to *NULL* to ignore the comment.

cdb_select_response() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_response()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cdb_select_result()**

PURPOSE: Selects and reads a VP results record from an image database file.

REFERENCE: System Software: Image Database

INCLUDE FILE: *cdb_defs.h, cdb_er.h*

USE:

```
char *cdb_select_result(fd, template, mode, length,
      sub_type, comment, class)
   int fd;
   cdb_record *template;
   int
      mode,
      *length,
      *sub_type;
   char **comment;
   int *class;
```

NOTES: **cdb_select_result()** returns a volatile pointer to the results data.

fd: is the file descriptor that identifies the target file.

template: points to the template record.

mode: is a bit field of values that can be bitwise OR'ed together for three types of purposes. When set, most of these bits specify particular components in the template record whose values you want to match in the candidate records. In addition, the first *mode* bit specifies the search direction in the target IDB file. The last 16 bits help to specify whether or not a **test_key()** function is called, and also allow you to define more modes within a user-defined index entry key.

In the names for the mode bit values, *SM* represents *Selection Mode*. The bit values and their meanings are:

CDB_SM_REV: Search in the *reverse* direction if this bit is set, or search forward if it is not set.

CDB_SM_SEQ: Match the *record sequence number* of the template record. Remember that record sequence numbers may not remain unique within an IDB file.

CDB_SM_TYPE: Match the record *type* of the template record.

CDB_SM_STYPE: Match the record *subtype* of the template record.

CDB_SM_DSIZE: Match the *data_size* component of the template record.

CDB_SM_COMMENT: Match the string that you specified in the *comment* component of the template record. The entire *comment* component in a candidate record is searched.

CDB_SM_KEY_MASK (field bits 16 through 31): Run your **test_key()** function to find a match for a *template key* (a record label) in the file index. You specify the template key in the *data* component of the template record. For a **cdb_select_*()** function to call **test_key()**, there are a variety of requirements that must be met. For complete details on those requirements, see the *Image Database* chapter of the *System Software* manual. Then, if *any* of the *CDB_SM_KEY_MASK* bits are set, **test_key()** is called and its test is run. You also can use these bits to define additional modes within the template key.

length: points to an output parameter. Upon return, it contains the byte size of the VP results data. Set this argument to *NULL* to ignore the length.

sub_type: points to an output parameter. Upon return, it contains the subtype of the results record. Set this argument to *NULL* to ignore the subtype.

comment: points to an output parameter. Upon return, it points to an ASCII comment associated with the results record. Set this argument to *NULL* to ignore the comment.

class: points to an output parameter. Upon return, it contains the grade of the VP results data. Set this argument to *NULL* to ignore the grade.

cdb_select_result() throws the following errors:

- *CDB_ERR_READ*, if there is a low-level software failure while reading.
- *CDB_ERR_EOF*, if **cdb_select_result()** reads forward beyond the end of a file or backward beyond the beginning of a file, or if no matching record is found.
- *CDB_ERR_MAGIC*, if the target IDB file is corrupted or was closed improperly. You may be able to recover some of the file. Try copying the file using **cdb_copy_file()** to generate a new file with a new index.
- *CDB_ERR_BADFD*, if *fd* does not identify an open IDB file.

FUNCTION: **cff_find()**

PURPOSE: Locates one or more strings within an image prior to reading or verification.

REFERENCE: Vision Tools: Reading

INCLUDE FILE: *cff.h*

USE:

```
cff_results *cff_find(image,font,results,
      contrast,fresults)
   cip_buffer *image;
   crv_font *font;
   int results;
   double contrast;
   cff_results *fresults;
```

NOTES: For each string it finds, **cff_find()** returns a y offset from the origin of the image and the height of a rectangle enclosing the string in a result record of type **cff_results**.

image: points to the **cip_buffer** of the image containing the string.

font: points to the font to which the characters belong.

results: the number of strings expected in the image. If you do not know the number of strings expected, set this number to a value large enough to accommodate the greatest number of strings that could possibly be in the image. If *result* is set to 0, **cff_find()** returns immediately.

contrast: used to eliminate invalid fields in cases where the number of fields found is greater than the number requested. If the contrast of a field is less than *contrast*, the field will not be considered as a potential string. If you can specify exactly the number of strings in the image, *contrast* will not influence the results.

fresults: points to the results returned. If *fresults* is NULL, a number of **cff_results** records equal to *results* is allocated and a pointer to the first result is returned. If *fresults* is non-NULL, you must allocate the **cff_results** records yourself. If you request more than one result, the results records are returned sorted in descending order according to how well they fit the string and font characteristics. If two or more strings equally fit the string and font characteristics, they are returned in descending order of character count.

cff_find() fills in the number of **cff_results** records specified by *results*. If the number of strings found is fewer than the number of records re quested, the unused **cff_results** records are returned with *y_offset* and *ht* set to 0.

STRUCTURE: **cff_results**

REFERENCE: Vision Tools: Reading

INCLUDE FILE: *cff.h*

DEFINITION:

```
typedef struct{
   int y_offset, ht;
}cff_results;
```

NOTES: The results structure returned by **cff_find()**.

y_offset: the y offset of the string from the origin of the image. The value returned for *y_offset* is as close to the top of the string as possible.

ht: the height (in pixels) of a rectangle enclosing the string plus some padding that is dependent on variation in the height of the models in the font.

STRUCTURE: **cfifo_buffer**

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

DEFINITION:

```
typedef struct{
   int
      *base, *end, *empty, *data,
      length, count,
      mask,
      error;
} cfifo_buffer;
```

NOTES: A FIFO buffer is implemented with a FIFO header and data is maintained in a ring buffer of integers that are allocated when the FIFO's header is allocated. A FIFO's header is a **cfifo_buffer** structure.

base: points to the beginning of the FIFO buffer (its base address).

end: points to the first address beyond the end of the FIFO, and is used to determine when the *data* and *empty* pointers must wrap around.

empty: points to the address of the next empty entry. Data is entered by writing it to the location pointed to by *empty,* and then incrementing the *empty* pointer (wrapping around if necessary). *empty* is decremented when data is read successfully. If *empty == data*, the buffer is empty. If *empty* points to the location immediately preceding *data,* the buffer is full.

data: points to the next data entry. Data is read by reading the location pointed to by *data,* and then incrementing the *data* pointer (wrapping around if necessary).

length: the total number of entries allowed. FIFOs are implemented as a ring buffer of *length+1* integers. One entry is always empty, so the buffer never contains more than *length* entries. The buffer is always one entry larger than *length* so that the *data* and *empty* pointers are not confused when the buffer is full or empty.

count: the number of filled entries in the FIFO. The read/get and write/put routines maintain *count.* A successful write operation increments *count;* a successful read decrements it. **cfifo_data()** checks to ensure that the states of *data* and *empty* are accurately reflected by *count* and returns *count* if this is the case. **cfifo_empty()** performs the same consistency check but returns the number of empty slots *(length - count)* if *count* is correct. If the *count* does not reflect the states of *data* and *empty*, **cfifo_empty()** throws the error *CFIFO_ERR_CHECK.*

mask: the parallel output line mask. *mask* determines the parallel output lines on which FIFO buffer data will be sent. It is used in the same fashion as **cio_write_out()**; a bit value of one indicates that the corresponding parallel output line (1-12 on a 2000/3000, 1-8 on a 4400, or 1 on a 4100/4200) is to be used; a bit value of zero indicates the line is to be masked. If no

data is present (underflow condition), the trigger action routine may assert *error* on the parallel output lines. *mask* and *error* provide a clean relationship with quick execution times for triggers, FIFOs, and output results.

error: the error code to assert when an error is detected. The only difference between the functions **cfifo_read()** and **cfifo_get()** and between the functions **cfifo_write()** and **cfifo_put()** is that **cfifo_get()** and **cfifo_put()** return *error* so that action routines can simply assert the error on parallel output lines rather than having to throw the error with **cct_throw()**.

The image acquisition software can use the parallel lines for triggering acquisition and strobes. You must ensure that direct use of the lines, use of the output FIFOs, and trigger use do not interfere with one another.

FUNCTION: **cfifo_data()**

PURPOSE: Returns the number of filled slots in a FIFO and performs a consistency check. It calls **cct_error()** if there is an inconsistency.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:

```
int cfifo_data(p_fifo)
   cfifo_buffer *p_fifo;
```

NOTES: If **cfifo_data()** is interrupted by any **cfifo_*** routine other than **cfifo_read()** operating on the same FIFO, it may not operate correctly.

p_fifo: points to the FIFO buffer on which to perform the consistency check and whose filled slots are to be counted.

The consistency check consists of the following:

- If the *empty* component of *p_fifo* is greater than or equal to the *data* component, the *count* component should equal *empty* minus *data*.
- If the *empty* component of *p_fifo* is less than the *data* component, the *count* component should equal the *end* component minus the *data* component, plus the *empty* component minus the *base* component.

cfifo_data() throws the error *CFIFO_ERR_CHECK* and resets the *count* component of *p_fifo* if the consistency check does not succeed.

FUNCTION: **cfifo_dump()**

PURPOSE: Displays the contents of a FIFO buffer.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:
```
void cfifo_dump(p_fifo);
   cfifo_buffer *p_fifo;
```

NOTES: *p_fifo:* points to the FIFO whose contents, including the *base*, *end*, *data*, and *empty* pointers, and the header, are to be displayed.

FUNCTION: **cfifo_empty()**

PURPOSE: Performs a consistency check on a FIFO and returns the number of unused slots.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:

```
int cfifo_empty(p_fifo)
   cfifo_buffer *p_fifo;
```

NOTES: **cfifo_empty()** may not operate properly if interrupted by any **cfifo_*** routine other than **cfifo_read()** operating on the same FIFO.

p_fifo: points to the FIFO buffer on which to perform the consistency check and whose unused slots are to be counted.

The consistency check consists of the following:

- If the *empty* component of *p_fifo* is greater than or equal to the *data* component, the *count* component should equal *empty* minus *data*.
- If the *empty* component of *p_fifo* is less than the *data* component, the *count* component should equal the *end* component minus the *data* component, plus the *empty* component minus the *base* component.

cfifo_empty() throws the error *CFIFO_ERR_CHECK* and resets the FIFO's *count* if the check does not succeed.

FUNCTION: **cfifo_free()**

PURPOSE: Calls the run time library function **free()** to deallocate a **cfifo_buffer**, then deallocates the **cfifo_header**.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:
```
void cfifo_free(p_fifo)
   cfifo_buffer *p_fifo;
```

NOTES: *p_fifo:* points to the **cfifo_buffer** to deallocate.

cfifo_free() throws any errors it encounters while returning the allocated space to the heap.

FUNCTION: **cfifo_get()**

PURPOSE: Reads data from the specified FIFO and returns the error value found in the **cfifo_buffer** if the buffer is empty.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:
```
int cfifo_get(p_fifo)
   cfifo_buffer *p_fifo;
```

NOTES: **cfifo_get()** can interrupt **cfifo_put()** and **cfifo_read()**; it can be interrupted by **cfifo_put()**.

p_fifo: points to the FIFO whose data is to be read.

FUNCTION: **cfifo_make()**

PURPOSE: Creates a **cfifo_buffer** structure and its associated ring buffer.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:

```
cfifo_buffer *cfifo_make(length,mask,error)
   int length, mask, error;
```

NOTES: *length:* the length of the buffer to create.

mask: the output mask for the FIFO.

error: the error to throw if there is a buffer underflow.

The ring buffer's header is initialized as follows:

- *base*, *empty*, and *data* point to the base of the ring buffer.
- *end* points to the address beyond the last usable address of the buffer (*base* + *length* + 1).
- *length*, *mask*, and *error* are set to the values you specify above.
- *count* is set to zero.

cfifo_make() throws the error *CVC_ERR_ALLOC* if there is not enough space on the heap to allocate the FIFO.

FUNCTION: **cfifo_put()**

PURPOSE: Writes data to a FIFO and returns the data.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:

```
int cfifo_put(p_fifo,data)
   cfifo_buffer *p_fifo;
   int data;
```

NOTES: If the FIFO is full, it returns the *error* value found in the error component of *p_fifo.*

p_fifo: points to the FIFO to which to write the data.

data: the data to be written to the FIFO.

cfifo_put() can interrupt **cfifo_get()** and **cfifo_read()**; it can be interrupted by **cfifo_get()**.

FUNCTION: **cfifo_read()**

PURPOSE: Reads data from a FIFO and calls **cct_error()** if the buffer is empty.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:
```
int cfifo_read(p_fifo)
   cfifo_buffer *p_fifo;
```

NOTES: **cfifo_read()** can be interrupted by **cfifo_put()** and **cfifo_get()**.

p_fifo: points to the FIFO from which to read. *p_fifo->data* is updated and *p_fifo->count* is decremented when you call this function. If incrementing *data* would exceed the buffer's bounds, *data* wraps around to the base of the buffer.

cfifo_read() throws the error *CFIFO_ERR_EMPTY* if the FIFO is empty.

FUNCTION: **cfifo_write()**

PURPOSE: Writes data to the FIFO and calls **cct_error()** if the buffer is full.

REFERENCE: System Software: Basic I/O

INCLUDE FILE: *cfifo.h*

USE:

```
void cfifo_write(p_fifo,data)
   cfifo_buffer *p_fifo;
   int data;
```

NOTES: **cfifo_write()** can be interrupted by **cfifo_get()**.

p_fifo: points to the FIFO to which to write.

data: points to the data to be written to the FIFO.

data is written to *p_fifo->empty*, the pointer to *p_fifo->empty* is updated, and the value of *p_fifo->count* is updated when you call this function. If incrementing *empty* would exceed the buffer's bounds, *empty* wraps around to the base of the buffer.

cfifo_write() throws the error *CFIFO_ERR_FULL* if the buffer is full.

FUNCTION: **cfocus_cleanup()**

PURPOSE: Deallocates the lookup table created by **cfocus_gradient().**

REFERENCE: Vision Tools: Auto-Focus

INCLUDE FILE: *cfocus.h*

USE: `void cfocus_cleanup()`

NOTES: Complete all calls to **cfocus_gradient()** before calling **cfocus_cleanup().**

FUNCTION: **cfocus_diff_sum()**

PURPOSE: Calculates the sum of intensity differences for an image.

REFERENCE: Vision Tools: Auto-Focus

INCLUDE FILE: *cfocus.h*

USE:
```
CIA_31_16 cfocus_diff_sum(image)
   cip_buffer *image;
```

NOTES: **cfocus_diff_sum()** returns an integer in **CIA_31_16** format. Values for images with a depth of 8 bits and an information content of 6 bits (two bit unused) range from 0.0 to 126.0. Values for images with a depth of 8 bits and an information content of 8 bits (only available with the Cognex 3000/4400) range from 0.0 to 510.0.

image: points to an image, which must be FAST8 or capable of being copied to FAST8.

If the image specified cannot be copied to a FAST8 image, or if it is NULL, the error *CGEN_ERR_BADARG* is thrown.

FUNCTION: **cfocus_gradient()**

PURPOSE: Calculates the average edge gradient for an image.

REFERENCE: Vision Tools: Auto-Focus

INCLUDE FILE: *cfocus.h*

USE:
```
CIA_31_16 cfocus_gradient(image)
   cip_buffer *image;
```

NOTES: **cfocus_gradient()** returns an integer in **CIA_31_16** format. Values range from 0.0 to 63.0. This range of values represents a conversion, using a logarithmic scale, of actual gradient calculations.

image: points to an image, which must be FAST8 or capable of being copied to FAST8.

The function creates a lookup table approximately 16K in size the first time it is called (see **cfocus_cleanup()**).

If the image specified cannot be copied to a FAST8 image, or if it is NULL, the error *CGEN_ERR_BADARG* is thrown.

FUNCTION: **cfocus_search()**

PURPOSE: Implements a search strategy for optimally locating the extrema of a function having one variable.

REFERENCE: Vision Tools: Auto-Focus

INCLUDE FILE: *cfocus.h*

USE:

```
CIA_31_16 cfocus_search(range,min_adjustment,evaluation,
      move)
   int range;
   int min_adjustment;
   c_function evaluation;
   c_function move;
```

NOTES: **cfocus_search()** returns an integer, in **CIA_31_16** format, indicating the final focus position. The focus position is relative to the search range.

range: an integer indicating the search range of the focusing task. Typically, the range should cover the entire range of your camera or focus ring. However, depending on your application, you may be able to restrict the search range and thus decrease the number of measurements necessary to find optimum focus. The unit of measure used for the *range* will be the same as used in your move function.

min_adjustment: indicates that the **cfocus_search()** function should terminate if a proposed focus position adjustment does not exceed this minimum amount. As optimum focus is approached, the distance between focus position adjustments gets increasingly smaller. In most cases, the last few focus position movements do not result in a significant improvement in focal quality. With the *min_adjustment* argument, you can eliminate insignificant focus position movements and the resultant image measurements and thus reduce the amount of time necessary to complete the Auto-Focus task.

min_adjustment assumes the same unit of measure as used in your move function. For example, if your move function defines camera movement in terms of motor steps, then specifying a *min_adjustment* of 3 indicates that **cfocus_search()** should terminate when a proposed movement is less than or equal to 3 motor steps.

The value of *min_adjustment* should be arrived at empirically. Observe the camera movements when *min_adjustment* is set to 0 to determine whether final position adjustments seem excessive. If so, observe focus quality using a variety of *min_adjustment* settings to arrive at the preferred setting.

evaluation: points to your evaluation function.

move: points to your move function. The move function must take one argument, which is the position to move to. The move function should return

TRUE if the move was legal, or should return NULL if the move was illegal (and therefore not performed).

CGEN_ERR_BADARG will be thrown if:

- *range* is 0 or a negative number
- *min_adjustment* is greater than or equal to *range*
- the *evaluation* or *move* is NULL

FUNCTION: **cfs_clear_boot()**

PURPOSE: Invalidates the default boot partition (the first boot partition on the bus) for the SCSI hard drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_clear_boot(msgs)
   int msgs;
```

NOTES: **cfs_clear_boot** has been superceded by two new functions: **cfs_clear_boot_hard** (to invalidate the default boot partition for the SCSI hard drive) and **cfs_clear_boot_floppy** (to invalidate the diskette in the SCSI floppy drive). **csf_clear_boot** (invalidates the default boot partition for the SCSI hard drive) is still supported to ensure upward compatibility for existing code. However, we recommend using **cfs_clear_boot_hard** and **cfs_clear_boot_floppy** for new code. This introduces a distinction between a hard drive and a floppy drive.

You invalidate the boot partition of a hard disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

msgs: specifies whether or not messages should be printed to the screen during execution. Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_clear_boot() returns 0 if successful or -1 if an error occurs during execution.

FUNCTION: **cfs_clear_boot_floppy()**

PURPOSE: Invalidates the disk in the SCSI floppy drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_clear_boot_floppy(msgs)
   int msgs;
```

NOTES: You invalidate a disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

msgs: is set to 1 to indicate that prompts and messages are sent to standard output (typically the screen). Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_clear_boot_floppy() returns 0 if successful or -1 if an error occurs during execution.

FUNCTION: **cfs_clear_boot_hard()**

PURPOSE: Invalidates the default boot partition (the first boot partition on the bus) for the SCSI hard drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_clear_boot_hard(msgs)
   int msgs;
```

NOTES: You invalidate the boot partition of a hard disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

msgs: specifies whether or not messages should be printed to the screen during execution. Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_clear_boot_hard() returns 0 if successful or -1 if an error occurs during execution.

FUNCTION: **cfs_clear_floppy()**

PURPOSE: Invalidates a bootable internal floppy diskette by clearing all the data from the disk.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_clear_floppy(msgs)
   int msgs;
```

NOTES: You invalidate a disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

msgs: specifies whether or not messages should be printed to the screen during execution. Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_clear_floppy() returns 0 if successful or -1 if there is an error during execution.

Note: Clearing the data includes removal of formatting information. The disk must be reformatted before it can be used again.

FUNCTION: **cfs_diskabort()**

PURPOSE: Forces a disk to close.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_diskabort(path)
   char *path;
```

NOTES: *path:* specifies the pathname of the disk to close.

If there is a serious disk error, you can force the disk to close with this function. When you call **cfs_diskabort()**, the disk information is removed from RAM, the count kept by **cfs_diskopen_count()** is set to zero, and any changes that were not written to the disk before the call to **cfs_diskabort()** are lost.

Caution: If you call **cfs_diskabort()** when a file is open for writing, the disk on which the file is located may be corrupted. You should call this function only as a last resort.

FUNCTION: **cfs_diskclose()**

PURPOSE: Closes a disk that has been opened with **cfs_diskopen()**.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_diskclose(path)
   char *path;
```

NOTES: *path:* points to the pathname of the disk to close. An error occurs if the specified disk has not been opened.

If the disk is open more than once, **cfs_diskclose()** simply decrements the count of the number of times the disk is open. If the disk is open only once, **cfs_diskclose()** closes the disk and clears the disk information from RAM. When a disk's information is cleared from RAM, its current working directory is reset to root.

cfs_diskclose() returns 0 if successful or -1 if there is an error during execution.

Caution: If you call **cfs_diskclose()** when a file is open for writing, the disk on which the file is located may be corrupted. You should call this function only as a last resort.

FUNCTION: **cfs_diskopen()**

PURPOSE: Opens a disk for use by the file system.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_diskopen(path)
   char *path;
```

NOTES: **cfs_diskopen()** opens a disk for use by the file system. More than one disk can be open at a time, and a disk can be opened more than once. You cannot perform any low level file system operations on a disk until it has been opened. A disk must be partitioned and formatted before it can be opened.

path: points to the pathname of the disk to open. If you specify a pathname that includes a file, this function opens the file after opening the disk.

cfs_diskopen() returns 0 if successful or -1 if there is an error during execution.

Note that the first time you open a disk, information about that disk is placed in RAM. The amount of space occupied by the disk information is 32 Kbytes for a 40 Mbyte disk. This number applies only to opening a disk. If you open files on a disk, more space will be required, depending on the sizes of the files.

The vision processor keeps a count of the number of times a disk has been opened. The disk information that is placed in RAM the first time you open a disk remains in RAM until the disk has been closed the same number of times it has been opened. Also, information about *each* open disk is stored in RAM. If you open more than one disk at a time, disk information could use a substantial amount of space. Be sure to close a disk with **cfs_diskclose()** when you are finished.

To open a file, you must first open the disk with the function **cfs_diskopen()**. To open multiple files, we recommend opening the disk each time with **cfs_diskopen()**. If the same disk name is supplied to each call to **cfs_diskopen()**, the same disk will be opened once for each call. Since multiple calls to **cfs_diskopen()** are allowed, you do not have to do any checking that a disk is already open. Remember to call **cfs_diskclose()** for each time **cfs_diskopen()** is called.

The first time you open a disk, disk parameters, such as the current working directory, are set to their default values. Any time the disk is subsequently opened the parameters are not changed. To be safe, you should explicitly set the parameters you require when using a disk.

cfs_diskopen() can open either a high or low density file system disk in a high density drive. Although a low density disk may be read in a high density drive, writing to a low density disk is not guaranteed to work.

FUNCTION: **cfs_diskopen_count()**

PURPOSE: Returns the number of times a disk has been opened.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_diskopen_count(path);
   char *path;
```

NOTES: *path:* points to the name of the disk whose open count you want returned. You may specify a complete pathname, but anything beyond the drive specification is ignored.

cfs_diskopen_count() returns -1 if there is an error during execution.

FUNCTION: **cfs_format()**

PURPOSE: Formats a disk.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_format(drive,msgs)
   char *drive;
   int msgs;
```

NOTES: **cfs_format()** has been superseded by **cfs_format_hard()** (to format a SCSI hard drive) and **cfs_format_floppy()** (to format a SCSI floppy drive). Cognex will continue to support **cfs_format()** to maintain upward compatibility. You should use **cfs_format_hard()** and **cfs_format_floppy()** for any new code that your write.

cfs_format() formats a disk. The disk may be:

- A 720 Kbyte 3.5 inch floppy disk in the vision processor's internal floppy drive (*a*) or in the SCSI floppy drive (*b*).
- An external hard disk (that has been partitioned) connected to the SCSI bus. The hard disk capacity after formatting depends on the size of the file system partition. It can be any size; however, if you install a hard disk with more than 8 Gbytes, only 8 Gbytes will be formatted. A hard disk must be partitioned before it can be formatted. Similarly, the hard disk must be formatted after partitioning.

drive points to the name of the drive to format. You may specify a complete pathname, but anything beyond the drive specification is ignored. *drive* must begin with either the internal floppy drive name (*a*), the SCSI floppy drive name (*b*), or an external hard drive, starting with *c*, followed by a colon.

msgs indicates whether or not the function should send prompts and messages to standard output (typically the screen). Specify 0 to prevent prompts and messages from being printed; specify 1 to allow them to be printed. If *msgs* is set to 1, messages are sent to standard output if an error occurs during execution.

Caution: No checking is performed to see if there is data on the specified disk when you call **cfs_format()**. This means that if there is any data on the disk, it will be overwritten *without warning*.

cfs_format() returns 0 if successful or -1 if an error occurs during execution.

cfs_format() throws the following errors:

- *CGEN_ERR_BADARG* if *drive* is not a legal identifier
- *CFS_ERR_ERROR* if there is an internal problem

FUNCTION: **cfs_format_floppy()**

PURPOSE: Formats a floppy disk in the internal floppy drive or the SCSI floppy drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_format_floppy(drive,flags,msgs)
   char *drive;
   int flags, msgs;
```

NOTES: **cfs_format_floppy()** formats a floppy disk. The disk may be:

- A 720 Kbyte 3.5 inch floppy disk in the vision processor's internal floppy drive (*a*) or in the SCSI floppy drive (*b*).
- A 1.4 Mbyte (high density) 3.5 inch floppy disk in the vision processor's internal floppy drive (*a*) or in the vision processor's SCSI floppy drive (*b*).

drive: points to the name of the drive to format. You may specify a complete pathname, but anything beyond the drive specification is ignored. *drive* must begin with either the internal floppy drive name (*a*) or the SCSI floppy drive name (*b*), followed by a colon.

flags: must be supplied with one of two arguments depending on the floppy disk you format:

- *CFS_LO_DENSITY* if you format a low density floppy in either the internal floppy drive (*a*) or the SCSI floppy drive (*b*).
- *CFS_HI_DENSITY* if you format a high density floppy in either the internal floppy drive (*a*) or the SCSI floppy drive (*b*).

For example, the following call formats the floppy disk in SCSI drive *(b)* as a high density disk:

```
cfs_format_floppy("b:",CFS_HI_DENSITY,1);
```

msgs: specifies whether or not messages should be printed to the screen during execution. Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

Caution: No checking is performed to see if there is data on the specified disk when you call **cfs_format_floppy()**. This means that if there is any data on the disk, it will be overwritten *without warning*.

cfs_format_floppy() returns 0 if successful or -1 if an error occurs during execution.

cfs_format_floppy() throws the following errors:

- *CGEN_ERR_BADARG* if *drive* is not a legal identifier

- *CFS_ERR_ERROR* if there is an internal problem
- *CFS_ERR_NOTIMP* if *flags* is set to *CFS_HI_DENSITY* when the drive is specified as drive (*a)*
- *CFS_ERR_NOTIMP* if *flags* is set to *CFS_HI_DENSITY*, the drive is specified as drive (*b*), but a low density disk is used

FUNCTION: **cfs_format_hard()**

PURPOSE: Formats the file system partitions on a hard disk for use with the file system software.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_format_hard(drive,flags,msgs)
   char *drive;
   int flags, msgs;
```

NOTES: The disk may be an external hard disk connected to the SCSI bus. The hard disk capacity after formatting depends on the size of the file system partition. It can be any size, however, if you install a hard disk with more than 8 Gbytes, only 8 Gbytes will be formatted. A hard disk must be partitioned before it can be formatted. Similarly, the hard disk must be formatted after partitioning.

Caution: No checking is performed to see if there is data on the specified disk when you call **cfs_format_hard()**. This means that if there is any data on the disk, it will be overwritten *without warning.*

drive: points to the name of the drive to format. You may specify a complete pathname, but anything beyond the drive specification is ignored. *drive* must begin with an external hard drive name, starting with *c*, followed by a colon.

For example, the following call formats the disk in external hard drive *c*:

```
cfs_format_hard("c:",0,0);
```

flags: reserved for future use, and must be set to 0.

msgs: is set to 1 to indicate that prompts and messages are sent to standard output (typically the screen). This allows messages to be printed if an error occurs during execution.

cfs_format_hard() returns 0 if successful or -1 if an error occurs during execution.

cfs_format_hard() throws the following errors:

- *CGEN_ERR_BADARG* if *drive* is not a legal identifier
- *CFS_ERR_ERROR* if there is an internal problem
- *CFS_ERR_NOTIMP* if *flags* is not set to 0

FUNCTION: **cfs_free_space()**

PURPOSE: Returns the total amount of free space available on the specified drive, in bytes.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
long cfs_free_space(drive)
   char *drive;
```

NOTES: *drive:* points to the disk to check.

Note: Since the value returned indicates the total amount of space left on the disk, not the largest contiguous block of space, the largest file you can save to the disk may be smaller than the value returned, because creating a file may use some disk space.

cfs_free_space() returns -1 if there is an error during execution.

FUNCTION: **cfs_get_case()**

PURPOSE: Returns the current setting of the case mapping enable/disable flag (1 indicates enabled, 0 indicates disabled). To change the mapping, call **cfs_set_case()**.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_get_case();
```

FUNCTION: **cfs_get_cwd()**

PURPOSE: Writes the name of the current working directory of the specified drive into the character buffer you specify. To change the current working directory, call **cfs_set_cwd()**.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_get_cwd(drive,buffer)
   char *drive;
   char buffer[CFS_NAMELENGTH];
```

NOTES: *drive:* points to the name of the drive whose CWD you want to know. You may specify a complete pathname, but anything beyond the drive specification is ignored.

buffer: is the character buffer that will hold the name of the current working directory. *buffer* must be of the size *CFS_NAMELENGTH* to ensure that it is large enough to hold the directory's entire pathname. The constant *CFS_NAMELENGTH* represents the longest supported pathname.

cfs_get_cwd() returns 0 if successful or -1 if there is an error during execution.

FUNCTION: **cfs_get_drive()**

PURPOSE: Fills the specified buffer with the name of the default drive. To change the default drive, call **cfs_set_drive()**.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
int *cfs_get_drive(buffer)
   char buffer[3];
```

NOTES: *buffer:* is the character buffer that will hold the name of the default drive. The name returned consists of the drive letter followed by a colon and a terminating NULL.

cfs_get_drive() returns 0 if successful or -1 if there is an error during execution.

FUNCTION: **cfs_get_separator()**

PURPOSE: Places the character currently being used as pathname separator, followed by a terminating NULL, into the specified buffer. To change the separator, call **cfs_set_separator()**.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
int cfs_get_separator(buffer);
   char *buffer;
```

NOTES: *buffer:* points to the buffer in which to place the character.

cfs_get_separator() returns 0 if successful or -1 if there is an error during execution.

FUNCTION: **cfs_gdone()**

PURPOSE: Deletes a **cfs_stat** structure and frees the memory allocated for it.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
void cfs_gdone(stat)
   cfs_stat *stat;
```

NOTES: *stat:* points to the **cfs_stat** structure used by **cfs_gnext()** (originally returned by **cfs_gfirst()**).

FUNCTION: **cfs_gfirst()**

PURPOSE: Allocates and fills in a **cfs_stat** structure.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
cfs_stat *cfs_gfirst(path)
   char *path;
```

NOTES: *path:* points to the pathname that specifies what to find. If you specify a wildcard, this function is called once for each file, directory, or volume whose name matches. As each name is returned, call **cfs_gnext()** to get the name following that name in the directory.

cfs_gfirst() allocates and fills in a **cfs_stat** structure. **cfs_gfirst()** fills in the structure with the first name that matches the specified pathname. If there are no wildcards, **cfs_gfirst()** will find the specified file or directory. If no name matches, **cfs_gfirst()** will fail.

cfs_gfirst() returns NULL if there is no match or if an error occurs during execution. You should check the value of **errno** if this function returns NULL in order to determine whether or not an error occurred.

FUNCTION: **cfs_gnext()**

PURPOSE: Finds the next name in a directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
cfs_stat *cfs_gnext(stat)
   cfs_stat *stat;
```

NOTES: *stat:* points to the **cfs_stat** structure returned by **cfs_gfirst()** or a previous call to **cfs_gnext()**.

cfs_gnext() returns NULL if there are no more matches, or if there is an error during execution. You should check the value of **errno** if this function returns NULL in order to determine whether or not an error occurred.

FUNCTION: **cfs_init()**

PURPOSE: Initializes file system globals and structures, determines what, if any, devices are connected to the SCSI port, and reads the partition table (the list of partitions) on every device it finds.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_init();
```

NOTES: Every time the vision processor is booted **cfs_init()** must be called, either explicitly or by a high level function, before any other low level file system function is called. **cfs_init()** also must be called after every call to **css_save_system()** or **csys_boot_ramsize()**.

Caution: Drive identifiers may change if you call **cfs_init()** after you add a disk, remove a disk, partition a new disk, or re-partition a disk in order to add or delete a file system partition. For example, if you have four initialized disks on the SCSI bus, then add a disk between the second and third disks, the third and fourth disk's identifiers will change the next time you call **cfs_init()**.

When **cfs_init()** is called, it checks all disks connected to the vision processor to see if a disk is open and fails a disk is open.

cfs_init() returns 0 if successful, less than 0 if an error occurs during execution, or greater than 0 if a disk is open.

FUNCTION: **cfs_mkdir()**

PURPOSE: Creates a new directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_mkdir(path)
   char *path;
```

NOTES: *path:* points to the pathname of the directory to create. The pathname can be a maximum of 200 characters long.

cfs_mkdir() returns 0 if successful or -1 if there is an error during execution.

FUNCTION: **cfs_partition()**

PURPOSE: Partitions an individual hard disk.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_partition(dev_num,units,boot_size,file_sys_size,
      msgs)
   int dev_num, units;
   unsigned long boot_size, file_sys_size;
   int msgs;
```

NOTES:

dev_num: the device number of the disk to partition (0-7).

units: specifies the number the logical unit to partition.

boot_size: specifies the number of sectors to be used for the boot partition. Specifying a *boot_size* of 0 indicates no boot partition. Specifying a *boot_size* of -1 indicates that all of the sectors not used for the boot partition will be used for the file system partition.

file_sys_size: specifies the number of sectors to be used for the file system partition. Specifying a *file_sys_size* of 0 indicates no file system partition. Specifying a *file_sys_size* of -1 indicates that all of the sectors not used for the boot partition will be used for the file system partition.

msgs: indicates whether or not the function should print prompts and messages on the screen. Specify zero to prevent prompts and messages from being printed; specify one to allow them to be printed.

Caution: Re-partitioning a disk overwrites everything on the disk. Also, if you partition a new disk, or re-partition an existing disk, the drive identifiers of existing drives may change. See **cfs_init()** for more information.**cfs_partition()** throws the following errors:

- *CGEN_ERR_BADARG* if the device or unit is not legal, a disk is open, or both *boot_size* and *file_sys_size* are -1
- *CFS_ERR_DEVUNIT* if the device number is not that of a hard disk
- *CFS_ERR_SCSI* if there is a hardware error

FUNCTION: **cfs_partition_all()**

PURPOSE: Creates partitions on all hard disks connected to the SCSI bus.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_partition_all(boot_size,msgs)
   int boot_size,msgs;
```

NOTES: *boot_size:* specifies whether or not to create a boot partition on drive *c* and, if so, the size, in *sectors* (1/2 Kbyte) of the partition to create. A *boot_size* of -1 indicates no boot partition, a *boot_size* of 0 indicates a default boot partition of 16 sectors (8 Mbytes), specifying any other value indicates the number of sectors to use for the boot partition.

When you reserve a boot partition, that section of the disk is not used for any other purpose. As a result, you can overwrite a saved system only by reformatting or re-positioning the disk, or by creating a new saved system.

msgs: indicates whether or not the function should print prompts and messages on the screen. Specify zero to prevent prompts and messages from being printed; specify one to allow them to be printed.

Caution: Re-partitioning a disk overwrites everything on the disk. Also, if you partition a new disk, or re-partition an existing disk, the drive identifiers of existing drives may change. See **cfs_init()** for more information.

cfs_partition_all() throws the following errors:

- *CGEN_ERR_BADARG* if the device or unit is not legal
- *CGEN_ERR_SCSI* if there is a hardware error

FUNCTION: **cfs_poll_disks()**

PURPOSE: **cfs_poll_disks()** polls all non-removable SCSI disks connected to the vision processor and returns the total number of sectors on each disk. This information may be useful for re-partitioning disks. Note that **cfs_poll_disks()** does not work with floppy disks: the only way to determine if a SCSI floppy drive is present is to put a disk into it and try to access it.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_poll_disks(size_array, msgs)
   unsigned long size_array[CFS_MAX_DEV][CFS_MAX_UNIT];
```

NOTES: *size_array:* a two dimensional array whose depth is determined by the maximum number of devices that may be connected to the vision processor, and whose width is determined by the maximum number of units into which each device may be divided. These two values are represented by the following two constants:

- *CFS_MAX_DEV,* the maximum total number of SCSI devices that may be connected on a SCSI bus (8)
- *CFS_MAX_UNIT,* the maximum number of units into which a SCSI device may be divided (8)

Note: You should use the constants instead of the corresponding integer values to ensure upward compatibility.

This function fills in each elements of *size_array*, corresponding to all possible SCSI devices and units. Each row in the array corresponds to one device. Each element in a row corresponds to a unit on that device. A zero in an element indicates that there is no unit of that number. Zeroes for all elements in a row indicate that either there is no device, or that the device is not a hard disk.

msgs: specifies whether or not messages should be printed to the screen during execution. Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_poll_disks() throws the error *CFS_ERR_SCSI* if a hardware error occurs.

FUNCTION: **cfs_rmdir()**

PURPOSE: Deletes an empty directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_rmdir(path)
   char *path;
```

NOTES: *path:* points to the pathname of the directory to remove.

If the directory contains any files or subdirectories, you must delete all of them before you can delete the directory. You delete files either with the C library function **remove()**, or with the high level file system function **cfs_util_delete()**.

cfs_rmdir() returns 0 if successful or -1 if there is an error during execution.

FUNCTION: **cfs_set_case()**

PURPOSE: Enables and disables the mapping of lowercase characters to uppercase characters.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
void cfs_set_case(flag)
   int flag;
```

NOTES: *flag:* specifies whether to enable or disable mapping. Set flag to one to enable mapping and to zero to disable mapping.

If you enable mapping, all filenames in the file system will be mapped to uppercase for use with MS-DOS; MS-DOS will be able to open all files created with the file system functions. If you disable mapping, MS-DOS can open only files whose names are uppercase; MS_DOS can list others, but cannot open them.

This function only affects whether or not files can be opened by MS-DOS; it does not affect whether files can be opened by a UNIX operating system.

FUNCTION: **cfs_set_cwd()**

PURPOSE: Sets a disk's current working directory to the specified directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_set_cwd(path)
   char *path;
```

NOTES: *path:* points to the pathname of the directory that will become the current working directory.

Caution: When you close a disk, the disk's information is cleared from RAM and the disk's CWD is reset to root.

cfs_set_cwd() returns 0 if successful or -1 if there is an error during execution.

FUNCTION: **cfs_set_drive()**

PURPOSE: Specifies which drive to use as the default drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_set_drive(drive)
   char *drive;
```

NOTES: *drive:* points to the name of the default drive. *drive* must begin with either the internal floppy drive name or an external hard drive name, followed by a colon. You may specify a complete pathname, but anything beyond the drive specification is ignored.

At start-up, the system sets the default drive to drive *c*. Use this function to change the default drive to a drive other than drive *c*. You may call **cfs_set_drive()** before calling **cfs_diskopen()**. Subsequent calls to low level functions will use the drive you specify with this function as the default drive.

cfs_set_drive() returns 0 if successful or -1 if there is an error during execution.

Note: If you change the default drive, then call a high level function or **cfs_init()**, the default drive will be changed back to drive *c*. As a result, you may need to reset the default drive after you call a high level function or **cfs_init()**.

FUNCTION: **cfs_set_separator()**

PURPOSE: Specifies the character to use between names in a pathnames.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_set_separator(string)
   char *string;
```

NOTES: You must use the same character in all pathnames. You may specify either a forward slash (/) or a backslash (\). The default is a forward slash.

string: points to the character to use. You specify the character in quotes, as shown (note that the backslash is quoted with a second backslash to ensure that it will be read in C code):

```
"/" or "\\"
```

cfs_set_separator() returns 0 if successful or -1 if there is an error during execution.

STRUCTURE: **cfs_stat**

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

DEFINITION:

```
typedef struct{
   char fname[9], fext[4];
   unsigned char fattribute;
   unsigned short ftime, fdate;
   unsigned long fsize;
}cfs_stat;
```

NOTES: Stores information about a file, directory, or volume.

fname and *fext:* the filename and extension, respectively. The arrays allocated for *fname* and *fext* include space for a terminating null character at the end of each string.

fattribute: specifies whether the name is that of a file, a directory, or a volume, using the following constants:

CFS_VOLUME, if the name is that of a volume

CFS_DIRENT, if the name is that of a directory

In a listing function, you determine whether a pathname is the name of a volume, a directory, or a file by using the bitwise AND operator on *fattribute* with these constants.

If the result of the AND operator on *fattribute* with *CFS_VOLUME* is true, the pathname is that of a volume. If the result of ANDing with *CFS_DIRENT* is true, the pathname is that of a directory. If neither AND is true, the pathname is that of a file (*fattribute* is NULL if the name is that of a file).

ftime and *fdate:* reserved for future use.

fsize: the size of the file in bytes. *fsize* is 0 for directories and volumes.

FUNCTION: **cfs_util_boot_to_file()**

PURPOSE: Reads a boot image from a boot partition and stores it to a file.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
void cfs_util_boot_to_file(dst)
   char *dst;
```

NOTES: *dst* specifies the name of the destination file.

cfs_util_boot_to_file() scans the SCSI bus, starting at drive 0, and copies the boot image from the first boot partition found into the destination file. The disk does not have to be open for the copy to occur.

cfs_util_boot_to_file() returns 0 if it executes properly and -1 if it fails.

cfs_util_boot_to_file() throws the error *CSS_ERR_NOPARTITION* if no boot partition is found.

FUNCTION: **cfs_util_clear_boot()**

PURPOSE: Invalidates the default boot partition (the first boot partition found when scanning the SCSI bus).

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_clear_boot();
```

NOTES: **cfs_util_clear_boot()** has been superseded by **cfs_util_clear_boot_hard()** (to invalidate the default boot partition for the SCSI hard drive) and **cfs_util_clear_boot_floppy()** (to invalidate the disk in the SCSI floppy drive). Cognex will continue to support **csf_util_clear_boot()** (to invalidate the default boot partition) to maintain upward compatibility. You should use **cfs_util_clear_boot_hard()** and **cfs_util_clear_boot_floppy()** for any new code you write.

```
void cfs_util_clear_boot();
```

cfs_util_clear_boot() invalidates the default boot partition (the first boot partition found when scanning the SCSI bus). You invalidate the boot partition of a disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

FUNCTION: **cfs_util_clear_boot_floppy()**

PURPOSE: Invalidates the disk in the SCSI floppy drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_util_clear_boot_floppy(msgs)
   int msgs;
```

NOTES: You invalidate a disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

msgs: is set to 1 to indicate that prompts and messages are sent to standard output (typically the screen). Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_util_clear_boot_floppy() returns 0 if successful or -1 if an error occurs during execution.

FUNCTION: **cfs_util_clear_boot_hard()**

PURPOSE: Invalidates the default boot partition (the first boot partition found when scanning the SCSI bus).

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
int cfs_util_clear_boot_hard(msgs)
    int msgs;
```

NOTES: You invalidate the boot partition of a disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**)

msgs: specifies whether or not messages should be printed to the screen during execution. Set *msgs* to 1 to enable messages; set it to 0 to disable messages.

cfs_util_clear_boot_hard() returns 0 if successful or -1 if an error occurs during execution.

FUNCTION: **cfs_util_clear_floppy()**

PURPOSE: Invalidates a bootable floppy disk by clearing all the data from the diskette.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
void cfs_util_clear_floppy();
```

NOTES: You invalidate a disk if you want to load a saved system from EPROM or over a serial line when you reboot the vision processor. (You must do this before running **sysgen**).

Note: Clearing the data includes removal of formatting information. The disk must be re-formatted before it can be used again.

FUNCTION: **cfs_util_copy()**

PURPOSE: Copies a source file to a destination file.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_copy(src, dst)
   char *src, dst;
```

NOTES: *src:* the name of the source file.

dst: the name of the destination file.

FUNCTION: **cfs_util_delete()**

PURPOSE: Deletes one or more files from a directory; it does not delete directories.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_delete(path)
   char *path;
```

NOTES: *path:* points to the pathname of the file to delete. If you specify a wildcard, *every* file whose name matches will be deleted.

Caution: When you call **cfs_util_delete()**, all files whose names match are deleted without warning.

FUNCTION: **cfs_util_file_to_boot()**

PURPOSE: Copies a boot image into the boot partition.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
void cfs_util_file_to_boot(src)
   char *src;
```

NOTES: *src* specifies the name of the source file containing the boot image.

The source file can be a saved system file or a file created by **cfs_util_boot_to_file()**. **cfs_util_file_to_boot()** scans the SCSI bus, starting at drive 0, and writes the boot image contained in the source file into the first boot partition found. The disk does not have to be open for the write to occur.

Caution: cfs_util_file_to_boot() does not check to see if data is already stored in the boot partition nor does it issue a warning if data is overwritten.

cfs_util_file_to_boot() returns 0 if it executes properly and -1 if it fails.

cfs_util_file_to_boot() throws the error *CSS_ERR_NOPARTITION* if no boot partition is found.

FUNCTION: **cfs_util_format()**

PURPOSE: Formats a disk.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_format(drive)
   char *drive;
```

NOTES: **cfs_util_format()** has been superseded by **cfs_util_format_hard()** (to format a SCSI hard drive), and **cfs_util_format_floppy()** (to format a SCSI floppy drive). Cognex will continue to support **cfs_util_format()** to maintain upward compatibility. You should use **cfs_util_format_hard()** and **cfs_util_format_floppy()** for any new code you write.

drive: points to the name of the drive to format. You may specify a complete pathname, but anything beyond the drive specification is ignored. *drive* must begin with either the internal floppy drive name (*a*), the SCSI floppy drive name (*b*), or an external hard drive, starting with *c*, followed by a colon.

For example, the following call formats the disk in external hard drive f:

```
cfs_util_format("f:",0);
```

The disk to format may be:

- A 720 Kbyte 3.5 inch floppy disk in the vision processor's internal floppy drive (*a*) or in the SCSI floppy drive (*b*).
- An external hard disk (that has been partitioned) connected to the SCSI bus. The hard disk capacity after formatting depends on the size of the file system partition. It can be any size; however, if you install a hard disk with more than 8 Gbytes, only 8 Gbytes will be formatted. A hard disk must be partitioned before it can be formatted. Similarly, the hard disk must be formatted after partitioning.

  Caution: No checking is performed to see if there is data on the specified disk when you call **cfs_util_format()**. This means that if there is any data on the disk, it will be overwritten *without warning*.

cfs_util_format() throws the following errors:

- *CGEN_ERR_BADARG* if *drive* is not a legal identifier
- *CFS_ERR_ERROR* if there is an internal problem

FUNCTION: **cfs_util_format_floppy()**

PURPOSE: Formats a floppy diskette in the internal floppy drive or the SCSI floppy drive.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_format_floppy(drive,flags)
   char *drive;
   int flags;
```

NOTES: **cfs_util_format_floppy()** formats a floppy disk. The disk may be:

- A 720 Kbyte 3.5 inch floppy disk in the vision processor's internal floppy drive (*a*) or in the SCSI floppy drive (*b*).
- A 1.4 Mbyte (high density) 3.5 inch floppy disk in the vision processor's internal floppy drive (*a*) or in the vision processor's SCSI floppy drive (*b*).

drive: points to the name of the drive to format. You may specify a complete pathname, but anything beyond the drive specification is ignored. *drive* must begin with either the internal floppy drive name (*a*) or the SCSI floppy drive name (*b*), followed by a colon.

flags: must be supplied with one of two values depending on the floppy disk you format:

- *CFS_LO_DENSITY* if you format a low density floppy in either the internal floppy drive (*a*) or the SCSI floppy drive (*b*).
- *CFS_HI_DENSITY* if you format a high density floppy in either the internal floppy drive (*a*) or the SCSI floppy drive (*b*)).

  For example, the following call formats the floppy disk in the internal floppy drive (*a*) as a high density disk:

  ```
  cfs_util_format_floppy("a:",CFS_HI_DENSITY);
  ```

High density floppy disks may be formatted only using high density drives and low density floppy disks may be formatted only using low density drives. If you attempt to format a high density disk using a low density drive or a low density disk using a high density drive, results are undefined.

Caution: No checking is performed to see if there is data on the specified disk when you call **cfs_util_format_floppy()**. This means that if there is any data on the disk, it will be overwritten *without warning*.

cfs_util_format_floppy() returns 0 if successful or -1 if there is an error during execution.

cfs_util_format_floppy() throws the following errors:

- *CGEN_ERR_BADARG* if *drive* is not a legal identifier

- *CFS_ERR_ERROR* if there is an internal problem
- *CFS_ERR_NOTIMP* if *flags* is set to *CFS_HI_DENSITY* when the drive is specified as drive *(a)*

FUNCTION: **cfs_util_format_hard()**

PURPOSE: Formats an external hard disk that is connected to the SCSI bus and has been partitioned. The hard disk capacity after formatting depends on the size of the file system partition. It can be any size, however, if you install a hard disk with more than 8 Gbytes, only 8 Gbytes will be formatted. A hard disk must be partitioned before it can be formatted. Similarly, the hard disk must be formatted after partitioning.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_format_hard(drive,flags)
   char *drive;
   int flags;
```

NOTES: The hard disk capacity after formatting depends on the size of the file system partition. It can be any size, however, if you install a hard disk with more than 8 Gbytes, only 8 Gbytes will be formatted. A hard disk must be partitioned before it can be formatted. Similarly, the hard disk must be formatted after partitioning

drive: points to the name of the drive to format. You may specify a complete pathname, but anything beyond the drive specification is ignored. *drive* must begin with the external hard drive name, starting with *c*, followed by a colon.

flags: reserved for future use, and must be set to 0.

cfs_util_format_hard() returns 0 if successful or -1 if there is an error during execution.

Caution: No checking is performed to see if there is data on the specified disk when you call **cfs_util_format_hard()**. This means that if there is any data on the disk, it will be overwritten *without warning*.

cfs_util_format_hard() throws the following errors:

- *CGEN_ERR_BADARG* if *drive* is not a legal identifier
- *CFS_ERR_ERROR* if there is an internal problem
- *CFS_ERR_NOTIMP* if *flags* is not set to 0

FUNCTION: **cfs_util_list()**

PURPOSE: Lists the files, directories, and volumes on a disk or the files and sub-directories in a directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:
```
void cfs_util_list(path)
   char *path;
```

NOTES: The listing includes the size, in bytes, of each file, directory, and volume; the number of files, directories, and volumes listed; and the amount of free space remaining on the disk, in bytes.

path: points to the pathname of the directory or drive to list. If you specify a filename with no wildcards, only that filename is listed.

FUNCTION: **cfs_util_mkdir()**

PURPOSE: Creates a new directory at the specified position in the file system tree.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_mkdir(path)
   char *path;
```

NOTES: This function does not support wildcard characters.

path: points to the pathname of the directory to create.

FUNCTION: **cfs_util_partition()**

PURPOSE: Creates partitions on an individual hard disk.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_partition(dev_num,unit)
   int dev_num, unit;
```

NOTES: A hard disk must be partitioned before it can be formatted. During execution this function prompts for information about how to partition the disk, including the sizes of partitions and whether or not to include a boot partition.

dev_num: the SCSI ID (0 to 6) of the disk to partition, where the drive at the lowest SCSI ID is 0, and the drive at the next SCSI ID is 1, and so on.

unit: the number of the logical unit to partition (0 to 7). For most hardware configurations, the unit is set to 0.

Caution: Re-partitioning a disk overwrites everything on the disk. Also, when you partition a new disk or re-partition an existing disk, the drive identifiers for other disks may change. If you create a file system partition, it must be formatted. Refer to *cfs_util_format_hard* for more information.

cfs_util_partition() throws the following errors:

- *CGEN_ERR_BADARG* if *device* and *unit* do not correspond to hard disks
- *CFS_ERR_SCSI* if there is a hardware error

FUNCTION: **cfs_util_partition_all()**

PURPOSE: Creates partitions on all hard disks connected to the SCSI bus.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_partition_all();
```

NOTES: During execution this function prompts for information about how to partition the disk, including the sizes of partitions and whether or not to include a boot partition.

Caution: Re-partitioning a disk overwrites everything on the disk. Also, when you partition a new disk, the drive identifiers for other disks may change. If you create a file system partition, it must be formatted. Refer to *cfs_util_format_hard* for more information.

cfs_util_partition_all() throws *CFS_ERR_SCSI* if there is a hardware error.

FUNCTION: **cfs_util_print_partition()**

PURPOSE: Prints to *stdout* the partition table for all hard disks attached to the SCSI bus.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_print_partition();
```

NOTES: The partition table contains information such as partition type, number of sectors, and starting sector.

The following is an example partition table.

```
Partition table for device 6, unit 0
   partition 0:Cognex boot partition, active flag 0x0;
      first sec 1, num sec 16384
   partition 1:MSDOS V4 huge partition, active flag 0x0;
      first sec 16385, num sec 85500
   partition 2:empty partition;type = 0
   partition 3:empty partition; type = 0
```

cfs_util_print_partition() throws the following errors:

- *CGEN_ERR_BADARG* if a disk drive is open
- *CFS_ERR_SCSI* if there is a hardware error

FUNCTION: **cfs_util_rename()**

PURPOSE: Changes the name of a file to a new name in the same directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_rename(old_path,new_name)
   char *old_path, *new_name;
```

NOTES: This function does not support wildcard characters.

old_path: points to the pathname of the file to rename.

new_name: points to the file's new name.

FUNCTION: **cfs_util_rmdir()**

PURPOSE: Deletes a directory.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_mkdir(path)
   char *path;
```

NOTES: You can delete a directory only if the directory contains no files or subdirectories. See **cfs_util_delete()**, above, for information about deleting files from a directory. **cfs_util_rmdir()** does not support wildcard characters.

path: points to the pathname of the directory to delete.

FUNCTION: **cfs_util_vol_to_boot()**

PURPOSE: Writes a *.vol* file into the boot partition.

REFERENCE: System Software: File System

INCLUDE FILE: *cfs_defs.h*

USE:

```
void cfs_util_vol_to_file(src)
   char *src;
```

NOTES: *src* specifies the name of the source file containing the *.vol* image. You must include the *.vol* extension in the file name.

The *.vol* file is created by *sysgen*. **cfs_util_vol_to_boot()** scans the SCSI bus, starting at drive 0, and writes the *.vol* image contained in the source file into the first boot partition found. The disk does not have to be open for the write to occur.

Caution: cfs_util_vol_to_boot() does not check to see if data is already stored in the boot partition nor does it issue a warning if data is overwritten.

cfs_util_vol_to_boot() returns 0 if it executes properly and -1 if it fails.

cfs_util_vol_to_boot() throws the error *CSS_ERR_NOPARTITION* if no boot partition is found.

STRUCTURE: **cgcl_calibration**

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

DEFINITION:

```
typedef struct{
   double
      resolution,
      center_x, center_y,
      focus_z,
      angle,
      p2o_angle;
   char x_sign, y_sign, rot_sign;
   cia_31_16 fov_x, fov_y;
   double
      fxx, fxy, fx0,
      fyx, fyy, fy0,
      faa, fa0,
      rxx, rxy, rx0,
      ryx, ryy, ry0,
      raa, ra0;
   char x_geom_res, y_geom_res;
   cgcl_geom **fwd, **rev;
   short width, height;
   bool fixed;
   c_function move, stable, release;
   caq_context *context;
   cip_buffer *window;
   cia_31_16 xo, yo, ao;
   double xp, yp, zp, ap;
   bool moving;
   cia_31_16 moving_to_xo, moving_to_yo;
   bool trap_op_to_image;
   int reserved[4];
}cgcl_calibration;
```

NOTES: **cgcl_calibration** contains all the calibration information for an inspection configuration. A pointer to this structure is returned by the function **cgcl_create_calibration()**.

resolution: the resolution of the object plane, in pixels per physical unit. This field is filled by the function **cgcl_create_calibration()**.

center_x, center_y: the physical coordinates that map to the center image pixel in a fixed camera system, as determined by **cgcl_create_calibration()**. In any other configuration, these fields are ignored.

focus_z: the physical *z* coordinate that corresponds to the focal plane of the camera, as determined by **cgcl_create_calibration()**.

angle: supplies the physical manipulator rotation angle that **cgcl_move()** passes on to your manipulator movement function. The initial value of this field is computed by **cgcl_create_calibration()** as follows:

$$a = m + rc$$

where:

- *a* is the initial value of the angle
- *m* is *cgcl_params.manipulator_a,* the angle of the manipulator in the manipulator's coordinate system, at calibration time.
- *r* is *cgcl_calibration.rot_sign,* which is computed at calibration time. It is either positive or negative 1, depending on whether the rotational "handedness" of the manipulator's coordinate system is as expected by the Calibration Tools. (The Calibration Tools expect positive manipulator rotation to cause counterclockwise rotation in the image.)
- *c* is the small angle of rotation between the calibration plate's *x* axis and the image's *x* axis, as determined by **cgcl_create_calibration()**.

If the objects you are inspecting are always handled in the same manner as the calibration plate at calibration time, you need not change the contents of this field. However, if you are inspecting objects that are to be picked up and rotated by the manipulator differently from the way the calibration plate was picked up and rotated before calibration, you must alter the value of this field before your manipulator rotates either the object or camera.

For example, if the object you are inspecting is to be rotated 90° from the calibration plate's orientation at calibration time before it is inspected, execute the following code, assuming your calibration structure is *cal_structure*:

```
cal_structure.angle += 90.0;
```

p2o_angle: the rotation between physical coordinates and object plane coordinates, in degrees. This value is set by **cgcl_create_calibration().**

x_sign and *y_sign:* used to create the coefficients of the linear equations that map between physical coordinates and the object plane). They are set to positive or negative 1 by the calibration function and are used as scalars to compensate for "handedness" differences between the object plane and physical coordinates. These fields are both set by **cgcl_create_calibration()**.

rot_sign: set to -1 if **cgcl_create_calibration()** learns that positive manipulator rotation is in the opposite direction than expected, otherwise it is set to 1. The Calibration Tools expect positive manipulator rotation to cause counterclockwise rotation in the image. This is set by **cgcl_create_calibration()**.

fov_x and *fov_y:* used internally as offsets to the current field of view.

fxx, fxy, fx0, fyx, fyy and *fy0:* the coefficients of the linear equation that transforms physical coordinates to object plane coordinates. They are set by **cgcl_create_calibration()**. If *(px, py)* defines a physical point, the object plane coordinates (*ox,oy*) are computed as follows:

$$ox = (px \cdot fxx) + (py \cdot fxy) + fx0$$

$$oy = (px \cdot fyx) + (py \cdot fyy) + fy0$$

faa and *fa0:* the coefficients of the linear equation that transforms the angle of orientation of a point from physical angle to object plane angle. If *pa* is a physical angle, the object plane angle is computed as follows:

$$oa = (pa \cdot faa) + fa0$$

rxx, rxy, rx0, ryx, ryy and *ry0:* the coefficients of the linear equation that transforms object plane coordinates to physical coordinates. If *(ox, oy)* defines a point in object plane coordinates, the physical coordinates (*px,py*) are computed as follows:

$$px = (ox \cdot rxx) + (oy \cdot rxy) + rx0$$

$$py = (ox \cdot ryx) + (oy \cdot ryy) + ry0$$

raa and *ra0:* the coefficients of the linear equation that transforms the angle of orientation of a point from object plane angle to physical angle. If *oa* is an object plane angle, the angle in the physical plane is computed as follows:

$$pa = oa \cdot raa + ra0$$

x_geom_res and *y_geom_res:* related to the dimensions of the *fwd* and *rev* tables as follows:

$$width = \frac{w + 2^{xgr} - 1}{2^{xgr}}$$

$$height = \frac{h + 2^{ygr} - 1}{2^{ygr}}$$

where *w* is the window width, *h* is the window height, *xgr* is the value of *x_geom_res* and *ygr* is the value of *y_geom_res.*

fwd: points to the distortion correction table that maps from the object plane to the image. It is computed by **cgcl_create_calibration().** The type **cgcl_geom** implements a table that maps a portion of the object plane to the image or a portion of the image to the object plane. It is defined internally.

rev: points to the distortion correction table that maps from the image to the object plane. It is computed by **cgcl_create_calibration().** The type **cgcl_geom** implements a table that maps a portion of the object plane to the image or a portion of the image to the object plane. It is defined internally.

width and *height:* the dimensions of *window* (see above).

fixed: TRUE if the camera is fixed, FALSE if the camera is attached to a manipulator. This field is copied from the *fixed* field of the **cgcl_params** structure.

move: if non-NULL, points to a function that moves the manipulator. A NULL argument indicates that there is no manipulator. This field is copied from the *move* field of the **cgcl_params** structure.

stable: if non-NULL, it points to a function that returns TRUE when the manipulator is stable and also returns the current physical coordinates of the manipulator. This function should be present if a *move* function is present. This field is copied from the *stable* field of the **cgcl_params** structure.

release: if non-NULL, it points to a function that tells the manipulator's controller that the manipulator is available. This function should be present if a *move* function is present. This field is copied from the *release* field of the **cgcl_params** structure.

context: points to the image acquisition context for the inspection system. It is the same value as the *context* parameter of **cgcl_create_calibration()**.

window: points to a **cip_buffer** indicating the region of the acquired image which is to be inspected.

xo, yo, ao: the object plane coordinates that map to the central pixel of the image (x, y, and angle). These fields are updated by **cgcl_sync()**.

xp, yp, zp, ap: the physical coordinates (x, y, z, and angle) of the manipulator, as returned by the manipulator's controller. These fields are updated by **cgcl_sync()**.

moving: is set TRUE by the function **cgcl_move()** which sets the manipulator in motion, and FALSE by the function **cgcl_sync()** when the manipulator has stabilized after a move.

moving_to_xo: filled by the function **cgcl_move()**. If *moving* is TRUE, this field contains the destination object plane *x* coordinate. If *moving* is FALSE, the contents of this field are not relevant. This field is used by **cgcl_sync().**

moving_to_yo: filled by the function **cgcl_move()**. If *moving* is TRUE, this field contains the destination object plane *y* coordinate. If *moving* is FALSE, the contents of this field are not relevant. This field is used by **cgcl_sync().**

trap_op_to_image: how to handle an object plane to image transformation that maps to a pixel that is not in the image portion of **cia_31_16** address space. If this field is TRUE, the error is trapped and the error *CGCL_ERR_BAD_ARG* is thrown. If this field is FALSE, the error is not trapped. The default value of this field is FALSE.

If you are passing the image coordinates to a graphics routine you may not want to trap the error, since graphics routines may clip their output to the visible portion of the image coordinate system automatically. If you do want to trap the error, set this field to TRUE.

reserved: reserved for future use.

FUNCTION: **cgcl_check_align()**

PURPOSE: Determines the angle of rotation, in degrees, between the image and the calibration plate. It also measures the resolution, the number of pixels per physical unit, of your inspection setup, and the image coordinates of the calibration plate circle nearest the center of the image.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_check_align(image,params,center_x,
      center_y, resolution,angle)
   cip_buffer *image;
   cgcl_params *params;
   double *center_x,*center_y,*resolution,*angle;
```

NOTES: *image:* an image of the calibration plate.

params: points to a **cgcl_params** structure. **cgcl_check_align()** uses the following fields: *polarity*, *hole_size*, *x_spacing*, *y_spacing*, *flags*, and *color*. All other fields are ignored.

center_x: points to an output parameter that, on return, contains the image *x* coordinate of the calibration plate circle nearest the center of the image.

center_y: points to an output parameter that, on return, contains the image *y* coordinate of the calibration plate circle nearest the center of the image.

resolution: points to an output parameter that, on return, contains the number of image pixels per physical unit of your inspection setup.

angle: points to an output parameter that, upon return, contains the angle of rotation (in degrees) between the calibration plate and the image.

cgcl_check_align() throws the following errors:

- *CGCL_ERR_BADIMG*, if it is unable to process the image you pass to it
- *CGCL_ERR_LOGIC*, if it cannot continue for some other reason

FUNCTION: **cgcl_check_tilt()**

PURPOSE: Supplies the tilt between the image and the calibration plate, in the *x* and *y* directions.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_check_tilt(image,params,x_tilt,y_tilt)
   cip_buffer *image;
   cgcl_params *params;
   double *x_tilt,*y_tilt;
```

NOTES: The output parameters (*x_tilt,y_tilt*) form a vector. To correct the tilt, adjust the camera or inspection surface so that the image moves in the direction and relative magnitude of this vector.

image: an image of the calibration plate.

params: points to a **cgcl_params** structure. **cgcl_check_tilt()** uses the following fields: *polarity, hole_size, x_spacing, y_spacing, flags,* and *color.* All other fields are ignored.

x_tilt: points to an output parameter that, upon return, contains a signed real number representing the tilt of the calibration plate, in the positive or negative *x* direction.

y_tilt: points to an output parameter that, upon return, contains a signed real number representing the tilt of the calibration plate, in the positive or negative *y* direction.

cgcl_check_tilt() throws the following errors:

- *CGCL_ERR_BADIMG*, if it is unable to process the image you pass to it
- *CGCL_ERR_ANGLE*, if the angle of rotation of the calibration plate is too high
- *CGCL_ERR_LOGIC*, if it cannot continue for some other reason

FUNCTION: **cgcl_create_calibration()**

PURPOSE: Generates and returns a pointer to a **cgcl_calibration** structure containing all the calibration information for the inspection configuration.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
cgcl_calibration *cgcl_create_calibration(context,window,
      params)
   caq_context *context;
   cip_buffer *window;
   cgcl_params *params;
```

NOTES: This structure relates the three coordinate systems, physical, object plane, and image, and provides manipulator control, if necessary. You invoke **cgcl_create_calibration()** once you have focused the camera on the calibration plate and aligned the camera with the surface.

context: points to the Cognex image acquisition structure. **cgcl_create_calibration()** uses this structure to acquire several images of the calibration plate. All users of this calibration should use *context* to acquire images.

window: points to a **cip_buffer** which is a window of the image of the calibration plate captured by *context*. If you are using the whole frame grabber, this can be points to **caq_image**. You must do all inspections related to this calibration within *window*.

params: points to a structure that contains all other input parameters for **cgcl_create_calibration()**. Some fields of this structure are copied into the **cgcl_calibration** structure that is output by **cgcl_create_calibration().**

cgcl_create_calibration() throws the following errors:

- *CGCL_ERR_BADIMG*, if it is unable to process an image that it acquires
- *CGCL_ERR_ORIGIN*, if it cannot locate the origin of the calibration plate and if it cannot locate an error correction table if the angle of rotation of the calibration plate is too high
- *CGCL_ERR_ANGLE*, if the angle of rotation of the calibration plate is too high.
- *CGCL_ERR_LOGIC*, if it cannot continue for some other reason

FUNCTION: **cgcl_delete_calibration()**

PURPOSE: Frees all storage related to a **cgcl_calibration** data structure.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:
```
void cgcl_delete_calibration(cal)
   cgcl_calibration *cal;
```

NOTES: *cal:* points to the calibration structure to delete. All memory related to this structure is freed.

STRUCTURE: **cgcl_error_table**

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

DEFINITION:

```
typedef struct {
   short
      table_width, table_height,
      origin_i, origin_j;
   double scale;
   char *error_table;
   int reserved;
} cgcl_error_table;
```

NOTES: This data structure contains the error correction table. This table contains corrections for calibration plate manufacturing errors. It contains, for each filled circle (i,j) on a calibration plate, an *x* correction and a *y* correction. This table is passed to the function **cgcl_create_calibration()** as part of the **cgcl_params** data structure.

table_width: the number of columns of circles in the calibration plate.

table_height: the number of rows of circles in the calibration plate.

origin_i: the index of the column that contains the origin circle of the calibration plate. The range of indices is between 0 and *table_width* -1. (The origin circle of the calibration plate is indicated by two rectangular marks on the calibration plate.)

origin_j: the index of the row that contains the origin circle of the calibration plate. The range of indices is between 0 and *table_height* -1. (The origin circle of the calibration plate is indicated by two rectangular marks on the calibration plate.)

scale: the scale factor applied to each *x* or *y* correction in your error table.

error_table: a pointer to a table that you create that has the *x* and *y* correction for each filled circle on your calibration plate.

FUNCTION: **cgcl_fov_size()**

PURPOSE: Returns the number of object plane pixels above, below, to the left of, and to the right of the center of the effective field of view.

REFERENCE: Vision Tools: Calibration Tools

DEFINITION:

```
void cgcl_fov_size(cal,left,top,right,bottom)
   cgcl_calibration *cal;
   int *left,*right,*top,*bottom;
```

NOTES: *cal:* points to the calibration structure containing the object plane and field of view data. The contents of the structure are not modified.

left: points to an output parameter which will contain the number of object plane pixels from the center to the left edge of the effective field of view.

top: points to an output parameter which will contain the number of object plane pixels from the center to the top edge of the effective field of view.

right: points to an output parameter which will contain the number of object plane pixels from the center to the right edge of the effective field of view.

bottom: points to an output parameter which will contain the number of object plane pixels from the center to the bottom edge of the effective field of view.

FUNCTION: **cgcl_get_fov()**

PURPOSE: Returns the center of the current field of view stored in a calibration structure.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_get_fov(cal,fov_x,fov_y)
   cgcl_calibration *cal;
   CIA_31_16 *fov_x,*fov_y;
```

NOTES: The result is returned as the object plane coordinates. For any calibration, this function returns the object plane *x* and *y* coordinates that were most recently input to the function **cgcl_set_fov()**.

cal: points to the calibration structure containing the object plane and field of view data. The contents of the structure are not modified.

fov_x: points to an output parameter. Upon return, it will contain the object plane *x* coordinate centered in the effective field of view.

fov_y: points to an output parameter. Upon return, it will contain the object plane *y* coordinate centered in the effective field of view.

FUNCTION: **cgcl_image_to_op()**

PURPOSE: Maps image coordinates to object plane coordinates.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_image_to_op(cal,xi,yi,ai,xo,yo,ao)
   cgcl_calibration *cal;
   CIA_31_16 xi,yi,ai;
   CIA_31_16 *xo,*yo,*ao;
```

NOTES:

cal: points to the calibration structure containing data needed by this function: the image to object plane transformation tables, along with the effective field of view information. The contents of this structure are not modified.

xi: the image *x* coordinate.

yi: the image *y* coordinate.

ai: the angle of orientation of the point (*xi,yi*). This argument is only relevant if the argument *ao* is not NULL.

xo: points to an output parameter. On return, it will contain the object plane *x* coordinate.

yo: points to an output parameter. On return, it will contain the object plane *y* coordinate.

ao: points to an output parameter. On return, it will contain the angle of orientation of the point (*xo,yo*). Set this parameter to NULL if you do not want an angle transformation.

FUNCTION: **cgcl_move()**

PURPOSE: Moves the manipulator to a new position, using the move-manipulator function that you supplied at calibration time.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_move(cal,xo,yo,height)
   cgcl_calibration *cal;
   CIA_31_16 xo,yo;
   double height;
```

NOTES: **cgcl_move()** centers the given object plane coordinates in the image and moves the manipulator a given distance from the focal plane.

cgcl_move() does not change the effective field of view data in the **cgcl_calibration** structure. To update all functions to the new field of view, use the function **cgcl_sync()** which waits until the manipulator is stable and then calls **cgcl_set_fov()**.

cal: points to the calibration structure containing the object plane tables and field of view information. This function sets the *moving* field to TRUE and stores the input values *xo* and *yo* in the fields *moving_to_xo* and *moving_to_yo*.

xo, yo: the target object plane coordinates. The manipulator moves so that the central pixel of the image aligns with these coordinates.

height: the physical distance from the focal plane to which the manipulator will be moved by **cgcl_move()**.

FUNCTION: **cgcl_op_to_image()**

PURPOSE: Maps object plane coordinates to image coordinates

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_op_to_image(cal,xo,yo,ao,xi,yi,ai)
   cgcl_calibration *cal;
   CIA_31_16 xo,yo,ao;
   CIA_31_16 *xi,*yi,*ai;
```

NOTES:

cal: points to the calibration structure containing data needed by this function: the object plane to image transformation tables, along with the effective field of view information. The contents of this structure are not modified.

xo: the object plane *x* coordinate.

yo: the object plane *y* coordinate.

ao: the angle of orientation of the point (*xo,yo*). This argument is only relevant if the argument *ao* is not NULL.

xi: points to an output parameter. On return, it will contain the image *x* coordinate.

yi: points to an output parameter. On return, it will contain the image *y* coordinate.

ai: points to an output parameter. On return, it will contain the angle of orientation of the point (*xi,yi*). Set this parameter to NULL if you do not want an angle transformation.

FUNCTION: **cgcl_op_to_physical()**

PURPOSE: Maps object plane coordinates to physical coordinates.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_op_to_physical(cal,xo,yo,ao,xp,yp,ap)
   cgcl_calibration *cal;
   CIA_31_16 xo,yo,ao;
   double *xp,*yp,*ap;
```

NOTES:

cal: points to the calibration structure containing data needed by this function: the object plane to physical coordinates transformation data, along with the effective field of view information. The contents of this structure are not modified.

xo: the object plane *x* coordinate.

yo: the object plane *y* coordinate.

ao: the angle of orientation of the point (*xo,yo*). This argument is only relevant if the argument *ap* is not NULL.

xp: points to an output parameter. On return, it will contain the physical *x* coordinate.

yp: points to an output parameter. On return, it will contain the physical *y* coordinate.

ap: points to an output parameter. On return, it will contain the angle of orientation of the point (*xp,yp*). Set this parameter to NULL if you do not want an angle transformation.

STRUCTURE: **cgcl_params**

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

DEFINITION:

```
typedef struct{
   int polarity;
   double
      hole_size,
      x_spacing, y_spacing,
      height;
   c_function move, stable, release;
   bool fixed;
   double
      manipulator_x, manipulator_y, manipulator_z,
      manipulator_a,
      max_rotation,
      origin_x, origin_y;
   cgcl_error_table *error_table;
   int flags,color;
} cgcl_params;
```

NOTES: **cgcl_params** contains input parameters for **cgcl_create_calibration(), cgcl_check_align()**, and **cgcl_check_tilt().** Those fields that are only used by **cgcl_create_calibration()** are indicated. All values representing physical units must use the same unit. Physical angles must be represented in degrees.

polarity: zero if the image of the calibration plate contains dark filled circles on a light background, one if the image of the calibration plate contains light filled circles on a dark background. The default value is zero.

hole_size: the diameter of each filled circle on the calibration plate, in physical units.

x_spacing: the *x* spacing between centers of adjacent circles on the calibration plate, in physical units.

y_spacing: the *y* spacing between centers of adjacent circles on the calibration plate, in physical units.

height: the thickness, in physical units, of your calibration plate. This field is only used by the Calibration Tools if your manipulator moves in the *z* direction. This field is only used by **cgcl_create_calibration()**.

move: a pointer to a move-manipulator function that you supply (if you are using a manipulator). If you have no manipulator, you must make this entry NULL. This field is only used by **cgcl_create_calibration()**.

stable: a pointer to a function that you supply, if you are using a manipulator. This function returns TRUE when the manipulator is stable after a move. It should also return, in its output parameters, the physical coordinates and

angle of rotation that the manipulator has achieved. If you have no manipulator, this field is ignored. This field is only used by **cgcl_create_calibration()**.

release: a pointer to a function that you supply, if you are using a manipulator. This function tells the manipulator's controller that the manipulator is no longer in use. If you have no manipulator, this field is ignored. This field is only used by **cgcl_create_calibration()**.

fixed: is TRUE if the camera is fixed, FALSE if the camera is attached to a manipulator. This field is only used by **cgcl_create_calibration()**.

manipulator_x, manipulator_y, and *manipulator_z:* depend on the inspection configuration.

If you are calibrating a system with a manipulator, these are the physical coordinates of the manipulator at calibration time, according to the coordinate system of the manipulator's controller. If you have set these values correctly at calibration time, invoking your move-manipulator function with these values as arguments should cause no movement.

If you are calibrating a system with no manipulator, assign these fields the physical *x, y,* and *z* coordinates that map to the center pixel of the image.

These fields are only used by **cgcl_create_calibration()**.

manipulator_a: provides the initial rotation of the manipulator at calibration time, in degrees. This is a reference rotation against which other rotations can be measured. This field is only used by **cgcl_create_calibration()**.

If you have a manipulator, *manipulator_a* should be the angle of the manipulator, according to the manipulator's controller. If you have set this value correctly at calibration time, invoking your move-manipulator function with this value as the *angle* argument should cause no manipulator rotation.

max_rotation: the maximum positive angle, in degrees, to rotate the manipulator during calibration. The default value is 90.0. Lower values can be used, but they will adversely affect the accuracy of manipulator calibration. This field is only used by **cgcl_create_calibration()**.

origin_x: required only if *fixed* (above) is FALSE. In that case, this field should contain the physical *x* coordinate of the origin of the calibration plate. This field is only used by **cgcl_create_calibration()**.

origin_y: required only if *fixed* (above) is FALSE. In that case, this field should contain the physical *y* coordinate of the origin of the calibration plate. This field is only used by **cgcl_create_calibration()**.

error_table: a pointer to a **cgcl_error_table** that contains, for each filled circle (i,j) on your calibration plate, an *x* correction and a *y* correction. If you have no error table this field should be NULL. This field is only used by **cgcl_create_calibration()**.

flags: contains flag settings that activate and control the diagnostic display. Two bits are defined:

- *CGCL_DEBUG:* If set, the Calibration Tools display diagnostic information on the screen and at the console during **cgcl_check_align()**, **cgcl_check_tilt()**, and **cgcl_create_calibration()**. This only occurs if you have included the file *gcl_diag.v*. By default, this flag is not set.

- *CGCL_PAUSE:* relevant only when *CGCL_DEBUG* is set. If *CGCL_PAUSE* is set, **cgcl_check_align()**, **cgcl_check_tilt()**, and **cgcl_create_calibration()** pause after displaying diagnostic information. The functions continue operation only when you press any trackball button. If this flag is not set when *CGCL_DEBUG* is set, there is only a brief delay after a diagnostic display. By default this flag is not set. To use an input device other than the trackball, change the function **cgcl_pause()**, in the file *gcl_diag.v*.

color: relevant only when *CGCL_DEBUG* is set in the *flags* field. It contains the color in which diagnostic information is displayed. This field can be set to either *CAQ_BLACK*, *CAQ_WHITE*, or *CAQ_BLINK* (the default).

FUNCTION: **cgcl_physical_to_op()**

PURPOSE: Maps physical input coordinates to object plane coordinates.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_physical_to_op(cal,xp,yp,ap,xo,yo,ao)
   cgcl_calibration *cal;
   double xp,yp,ap;
   CIA_31_16 *xo,*yo,*ao;
```

NOTES:

cal: points to the calibration structure containing data needed by this function: the physical coordinates to object plane transformation data, along with the effective field of view information. The contents of this structure are not modified.

xp: the physical *x* coordinate.

yp: the physical *y* coordinate.

ap: the orientation angle of the point (*xp,yp*). (This argument is only relevant if the argument *ao* is not NULL.)

xo: points to an output parameter. On return, it will contain the object plane *x* coordinate.

yo: points to an output parameter. On return, it will contain the object plane *y* coordinate.

ao: points to an output parameter. On return, it will contain the angle of orientation of the object plane point (*xo,yo*). Set this parameter to NULL if you do not want an angle transformation.

FUNCTION: **cgcl_receive()**

PURPOSE: Retrieves a **cgcl_calibration** structure from a storage location, such as a host computer or NVM.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
cgcl_calibration *cgcl_receive(name_p,input_fn,
      input_params)
   char *name_p;
   char *(*input_fn)();
   char *input_params;
```

NOTES: *name_p:* passed to the I/O function during initialization.

input_fn: points to a function to get the calibration structure from the hardware. You may write your own input function, or use one of the Cognex input functions.

input_params: points to a parameters structure to be sent to the input function. If you are using a Cognex input function, then the input parameters are specified by a parameters structure specific to the type of transfer taking place.

If you are using your own input function, you can pass data to this function with the *input_params* pointer.

FUNCTION: **cgcl_set_fov()**

PURPOSE: Establishes a new relationship between physical coordinates, the object plane and the image by altering the calibration structure.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_set_fov(cal,new_fov_x,new_fov_y)
   cgcl_calibration *cal;
   CIA_31_16 new_fov_x,new_fov_y;
```

NOTES: The input object plane coordinates are stored as the center of the effective field of view and all fields that depend on these coordinates are updated. The input coordinates need not reflect the position of the camera. However, if you want the calibration structure to reflect the position of the camera after a move, you should invoke **cgcl_sync()** which invokes **cgcl_set_fov()**. You must invoke this function to initialize a calibration structure before you can use any of the coordinate transformation functions.

Once this function has been invoked, old field of view information stored in the calibration structure is lost. If you need to save the previous field of view information, you can do this by invoking **cgcl_get_fov()** before you invoke **cgcl_set_fov().**

cal: a pointer to the calibration structure containing the object plane and field of view data. The following fields are updated: *fov_x, fov_y, fxx, fxy, fx0, fyx, fyy, fy0, faa, fa0, rxx, rxy, ry0, ryx, ryy, ry0, raa,* and *ra0.*

new_fov_x: the object plane *x* coordinate to be centered in the field of view.

new_fov_y: the object plane *y* coordinate to be centered in the field of view.

FUNCTION: **cgcl_sync()**

PURPOSE: Synchronizes the calibration structure with the position of the manipulator.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
void cgcl_sync(cal)
   cgcl_calibration *cal;
```

NOTES: When you call **cgcl_sync()**, all field of view information is lost. If you call **cgcl_sync()** while the manipulator is moving, it waits for the manipulator to stabilize before calculating the new field of view. This function calls **cgcl_set_fov()**.

cal: a pointer to the calibration structure containing the object plane tables and field of view information. This function sets the *moving* field to FALSE when the manipulator has stabilized. Since this function calls **cgcl_set_fov()**, all calibration structure fields modified by **cgcl_set_fov()** are changed.

cgcl_sync() throws the following error:

- *CGCL_ERR_SYNC* if no corresponding **cgcl_move()** call has been made

FUNCTION: **cgcl_transmit()**

PURPOSE: Transfers a **cgcl_calibration** structure to a specified location.

REFERENCE: Vision Tools: Calibration Tools

INCLUDE FILE: *cgcl.h*

USE:

```
int cgcl_transmit(cal,name_p,xfer_params,output_fn,
      output_params)
   cgcl_calibration *cal;
   char *name_p;
   cio_xfer_params *xfer_params;
   c_function output_fn;
   char *output_params;
```

NOTES: You can specify a number of transmission options, such as type of data compression, means of transmission, and the format in which to transmit the data. Depending on your choice of options, you can optimize transfers for speed or storage requirements. (For this release, the only type of data compression available is *packed*.) **cgcl_transmit()** is independent of any particular hardware device. You specify a hardware device by passing a pointer to a specific I/O function.

Note: Transfers over parallel I/O lines require the use of the Cognex I/O Option Board.

cal: points to the **cgcl_calibration** structure to transmit.

name_p: passed to the I/O function during initialization.

xfer_params: points to transfer parameters. These parameters are specified by the **cio_xfer_params** data structure and define such things as compression style, data transfer formats, and ASCII line termination.

If you specify NULL for *xfer_params*, the following defaults are used:

cio_xfer_params.style = CIO_XFER_STYLE_PACKED
cio_xfer_params.prep = CIO_XFER_PREP_ASCII
cio_xfer_params.ascii.chars_per_line = 78
cio_xfer_params.ascii.ending = CIO_XFER_ENDING_LF

output_fn: points to a function to send the **cgcl_calibration** structure to the hardware. You can write your own output function, or use one of the Cognex output functions.

If you specify NULL for the *output_fn* argument, the system returns a byte count without actually transmitting any data.

Note: If you specify NULL for *output_fn* and if your output function transmits more characters than it receives the count returned will be incorrect. The Cognex output functions do not transmit more characters than they receive. In general, it is best to let the output function return the byte count

by specifying the flag *CIO_XFER_GEN_FLAG_COUNT_ONLY* in the I/O parameters structure.

output_params: points to parameters to be sent to the output function. If you are using a Cognex output function, then the output parameters are specified by a parameters structure specific to the type of transfer taking place.

If you are using your own output function, you can pass data to it with the *output_params* pointer.

cgcl_transmit() throws the error *CGEN_ERR_NOTIMP* if the transfer style in the **cio_xfer_params** structure is set to *CIO_XFER_STYLE_RUN_LENGTH* or *CIO_XFER_STYLE_COMPRESS*.

FUNCTION: **cght_delete_model()**

PURPOSE: Frees all memory associated with a model.

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

USE:
```
void cght_delete_model(model)
   cght_model *model;
```

NOTES: *image:* points to the model to delete.

cght_delete_model() throws the following error:

- *CGHT_ERR_BADARG* if *model* is not a valid pointer to a model

FUNCTION: **cght_display_model()**

PURPOSE: Displays a contour finding model in an image.

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

USE:

```
cip_buffer *cght_display_model(model,image,x0,y0,mag,
      arm,color)
   cght_model *model;
   cip_buffer *image;
   int x0, y0, mag, arm, color;
```

NOTES: A contour finding model is characterized by its edges. **cght_display_model()** displays the model's edge magnitude image. The origin of the model, as set in the **cght_model** structure, is displayed as a cross.

model: points to the model to display.

image: points to the image that will contain the model.

x0, y0: the image coordinates at which to place the upper left corner of the model. If either value is -1, the model is centered in the image in that dimension.

mag: a magnification factor. **cght_display_model()** uses the image processing function **cip_expand()** to magnify the model image before displaying it.

arm: the length of each of the four arms that make up the cross used to display the origin.

color: the grey value of the cross marking the origin.

This function throws the error *CGHT_ERR_BADARG* if any of the following is true:

- The *CGHT_RTRN_MAG* flag is not set in the **cght_train_params** structure that is related to this model.
- The magnification factor is zero or negative.
- *model* doesn't point to a valid model, or *image* doesn't point to a valid image.
- The model doesn't fit into the image at the supplied *x* and *y* coordinates.

STRUCTURE: **cght_model**

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

DEFINITION:

```
typedef struct {
   int
      id,
      width, height,
      xres, yres;
   float x0, y0;
   int dir_cnt, edge_cnt, ram_size;
   cip_buffer *mag, *ang;
   int
      *dir_hist,
      angle_tol, scale_tol,
      peak_detect;
} cght_model;
```

NOTES: **cght_model** implements a model. It is generated by the function **cght_train()**. Except as noted, you may inspect these values but should not change them.

id: not initialized by **cght_train()**. It is a label field which you can use to identify this model or for any other purpose.

width, height: If *type* (below) is *CGHT_NORMAL*, these are the width and height of the training image in full resolution units. These values are not affected by the values *xres* a nd *yres.*

res, yres: the horizontal and vertical resolutions at which the model is trained. These values are copied from the *xres* and *yres* fields of the **cght_train_params** structure.

x0, y0: the horizontal and vertical coordinates of the origin of the model. These values can be positive or negative. They are the coordinates of the pixel in the model that will correspond to the runtime image coordinates returned by **cght_run().**

x0, and *y0 are* interpreted by **cght_run()** as distances, in image buffer coordinates, from the upper left corner of the model, which is the origin of the training image. They are both set to 0.0 by **cght_train()**, and you may change them if you want results relative to some other point in the model. For instance, if you change *x0, y0* to (1.0,1.0), **cght_run()** will return the point one pixel to the right of and below the upper left corner of the model.

dir_cnt: the number of angles that can be computed with the **cip_edge_data** structure that was used to create the model. This value is a function of the number of bits used to express angle. It is used by **cght_run()** to insure that the model matches the runtime parameters.

edge_cnt: the number of edge pixels found in the model. As this value increases, the speed of contour finding usually decreases.

ram_size: the size, in bytes, of the **cght_model** data structure and all tables created by **cght_train()**.

mag: points to the magnitude image created from the training image. This field is filled only if the *CGHT_RTRN_MAG* flag is set in the **cght_train_params** structure that is related to this model. The magnitude image is useful for tuning your model and for displaying it.

ang: points to the angle image created from the training image. This field is filled only if the *CGHT_RTRN_ANG* flag is set in the **cght_train_params** structure that is related to this model. The angle image is useful for designing and displaying your model.

dir_hist: points to a direction histogram created by **cght_train ().** This is a 360 element array of integers. If *dhist* is a direction histogram, then the value stored at *dhist[n]* is the number of edge pixels in the training image with angle *n.* This field is supplied as a convenience to the user and is filled only if the *CGHT_RTRN_DHIST* flag is set in the **cght_train_params** structure that is related to this model. A direction histogram is useful for comparing the runtime result with the model.

angle_tol: copied from the *angle_tol* field of the **cght_train_params** structure used to create this model.

scale_tol: not implemented.

peak_detect: copied from the *peak_detect* field of the **cght_train_params** structure used to create this model. When contour finding is run on a model made with a peak-detected image, the runtime image is also peak-detected.

STRUCTURE: **cght_params**

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

DEFINITION:

```
typedef struct {
   int
      results,
      accept,
      confusion,
      peak_sep,
      cover_pct;
   cip_edge_data *edge_data;
   int flags;
} cght_params;
```

NOTES: **cght_params** contains parameters to the contour finding function **cght_run()**.

results: the number of patterns to find.

accept: a value in the range 0 - 1000. It is an accept threshold: the minimum score that a pattern must achieve to be considered found.

confusion: a value in the range 0 - 1000. It is a confusion threshold: the maximum score that a pattern that is *not* an instance of the model pattern can achieve. You should set the confusion threshold higher than the accept threshold.

peak_sep: the peak separation, the minimum distance (in pixels) allowable between two patterns. No two patterns will be reported if they are closer than *peak_sep*. This field must be greater than or equal to zero.

If peak separation is *n*, then **cght_run()** will not find a feature unless it is either vertically or horizontally separated by *n* pixels from any other feature. Note that the condition must only be satisfied in one direction. (The distance between feature *a* and feature *b* is measured by placing each feature in a bounding box the size of the model and measuring from the upper left corner of the bounding box of feature *a* to the upper left corner of bounding box of feature *b*.)

Zero is a special case. If *peak_sep* is zero, **cght_run()** will not find two features unless they are either *w* pixels apart vertically or *h* pixels apart horizontally, where *w* is the width of the model and *h* is the height of the model. This value is the default setting for peak separation.

cover_pct: the cover percentage, the amount of a pattern that is only partially contained in the runtime image that must be present for the pattern to be found. The cover percentage is applied to each dimension of the model to determine the minimal amount of a pattern that must be present in the runtime image to be found. For instance if *cover_pct* is 75, and if the model has a horizontal dimension of 200 pixels and a vertical dimension of 100

pixels, then the amount of the pattern that must be present for the patterns to be found fits in a bounding box that is horizontally 150 pixels and vertically 75 pixels.

edge_data: points to an initialized **cip_edge_data** structure. This can be a different structure than the one used to train the model. However, the two structures must be identical with the following two exceptions:

- The *cip_edge_data.params.run_rej_pct* fields may differ.
- The *cip_edge_data.params.rtail_n* fields may differ.

flags: reserved for future use.

FUNCTION: **cght_receive_model()**

PURPOSE: Retrieves a contour finding model from a storage location, such as a host computer or nonvolatile memory. It returns a pointer to the model.

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

USE:

```
cght_model *cght_receive_model(name,input_func,
      input_params)
   char *name;
   c_functioninput_func;
   char *input_params;
```

NOTES: *name:* passed to the I/O function during initialization. This is usually used as a file name but may be NULL.

input_func: points to a function to receive the model from hardware. You can write your own function or use one of the Cognex input functions: **cio_par_receive()**, for receiving data over parallel lines, or **cio_ser_receive()**, for receiving data over a serial line.

If you are using your own input function, you can pass data to this function with the *input_params* pointer.

input_params: points to parameters to be passed to the input function. If *input_func* is **cio_par_receive()**, this should be a pointer to an initialized **cio_par_params** structure. If *input_func* is **cio_ser_receive()**, this should be a pointer to an initialized **cio_ser_params** structure.

cght_receive_model() throws the following errors:

- *CGHT_ERR_BADARG,* if any argument is not valid
- *CGHT_ERR_PARAMS,* if a field of a parameters structure is not valid

STRUCTURE: **cght_results**

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

DEFINITION:

```
typedef struct {
   int tag;
   char found;
   float x,y;
   short score;
   int time;
   cght_model *model;
   char *stats;
} cght_results;
```

NOTES: The **cght_results** structure is filled by the function **cght_run().** It contains the results for a single pattern in the runtime image. Results records are filled in for as many results as were requested in the **cght_params** data structure.

tag: a label field which you can use to identify this structure or for any other purpose.

found: a boolean field. If *score* is greater than or equal to the accept threshold *found* is TRUE, otherwise it is FALSE.

x,y: the horizontal and vertical coordinates of the origin of the pattern in the image. These are in full resolution coordinates, regardless of whether the model used was in reduced resolution. These values are only meaningful if *score* is greater than zero.

score: the score of the pattern. This measures the similarity between the pattern found and the model pattern; it is the number of edges that contribute to an instance of the pattern, as a fraction of the number of edges in the model, scaled to the range 0-1000.

time: the total time, in milliseconds, that **cght_run()** used to find all patterns in the image. If you have more than one **cght_results** structure, the only valid *time* field is the one in the first structure returned.

model: points to the model used to locate this pattern.

stats: reserved for internal use.

FUNCTION: **cght_run()**

PURPOSE: Return the highest score of any pattern resembling the model pattern in the runtime image. If you supply an array of results structures, **cght_run()** may find more than one pattern matching the pattern in the model.

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

USE:

```
int cght_run(image,model,params,results)
   cip_buffer *image;
   cght_model *model;
   cght_params *params;
   cght_results *results;
```

NOTES: *image:* points to the image in which to locate one or more patterns that match the model pattern.

model: points to the model of the pattern.

params: points to the runtime contour finding parameters. They include the number of patterns to find, the accept threshold, and the minimum separation between patterns.

result: points to an array of results structures. You must allocate as many results structures as you specify in the *cght_params.results* field. If not, **cght_run()** may overwrite memory.

cght_run() throws the following errors:

- *CGHT_ERR_PARAMS,* if there is bad data in the **cght_params** structure
- *CGHT_ERR_BADANG,* if the number of bits per angle in the model is different than the number of bits per angle in the runtime parameters structure
- *CGHT_ERR_BADARG,* if any of the arguments are not valid pointers

FUNCTION: **cght_train()**

PURPOSE: Trains a contour finding model.

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

USE:

```
cght_model *cght_train(image,mask,params)
   cip_buffer *image,*mask;
   cght_train_params *params;
```

NOTES: *image:* points to the image to train as a model. This image should be FAST8. If it is not, the model training function copies it to a FAST8 image temporarily. The image must have width less than or equal to 256 pixels. The height multiplied by the width must be less than or equal to 65,536 pixel.

mask: Not implemented. You must set this field to zero.

*params: p*oints to a training parameters structure. This structure contains information that will be used to create the model, such as angle and resolution.

cght_train() throws the following errors:

- *CGHT_ERR_PARAMS,* if there is bad data in the **cght_train_params** structure
- *CGHT_ERR_MSIZE,* if the width of the model image is not less than or equal to 256 pixels or if the height multiplied by the width is not less than or equal to 65,536 pixels
- *CGHT_ERR_BADARG,* if *model* doesn't point to a valid model or if *params* doesn't point to a valid **cght_train_params** structure

STRUCTURE: **cght_train_params**

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

DEFINITION:

```
typedef struct {
   int angle_tol, scale_tol, xres, yres;
   bool peak_detect;
   cip_edge_data *edge_data;
   int flags;
} cght_train_params;
```

NOTES: **cght_train_params** contains parameters to the model initialization functions, **cght_train()** and **cght_train_seg().**

angle_tol: the required tolerance to rotation of the model, in degrees. If *cght_train_params.angle_tol* is *n*, then the tolerance to rotation of the model will be in the range *-n°* to *+n°*. This value must be greater than or equal to zero. The amount of memory required to store the model increases as *angle_tol* increases.

scale_tol: not implemented.

xres: the horizontal spatial averaging factor of the model image. A value of zero or one indicates no change in horizontal resolution. If *xres* is *n*, the training image is spatially averaged horizontally at the resolution 1/*n* before the model is created. **cght_run()** spatially averages the runtime image to the same horizontal resolution as the training image before locating patterns. This value must be greater than or equal to zero. It must be zero if the model is being trained with the function **cght_train_seg()**.

yres: the vertical spatial averaging factor of the model image. A value of zero or one indicates no change in vertical resolution. If *yres* is *n*, the training image is spatially averaged vertically at the resolution 1/*n* before the model is created. **cght_run()** spatially averages the runtime image to the same vertical resolution as the training image before locating patterns. This value must be greater than or equal to zero. It must be zero if the model is being trained with the function **cght_train_seg()**.

peak_detect: a boolean field. If it is set to TRUE, **cght_train ()** will filter the training image with the function **cip_peak()** a nd build the model from the filtered image. Since this removes some edges from the model, it makes the model less discriminating but less susceptible to noise. Contour finding will be faster on a peak-detected model. If you do not want to filter your training image in this way, set this field to FALSE. This field is ignored by the function **cght_train_seg()**.

Note: When the contour finding function **cght_run()** is run on a model with *peak_detect* set, it filters the runtime image with **cip_peak()** before it locates features.

edge_data: points to an initialized **cip_edge_data** structure. This structure controls how edges are detected and how angles are computed during model training. The **cip_edge_params** structure contained in the **cip_edge_data** structure should be initialized in the following way:

- If the training image has been acquired on the Cognex 3000/4400, with an eight-bit acquisition, the *num_bits_pel* field must be eight, otherwise it must be six.
- The *num_bits_mag* field should be six (other values will be accepted, but will not improve performance).
- The *setup_rej_pct* field must be zero.

flags: a flag word. Three flags are defined. By setting them, you can request the following actions by **cght_train()**:

- *CGHT_RTRN_DHIST:* Make a direction histogram of the training image and put a pointer to it in the model data structure. A direction histogram is a table that assigns a value *n* to each edge angle θ if there are *n* edge pixels (above a given magnitude threshold) whose edge angle is θ. It is useful for comparing the runtime result with the model.
- *CGHT_RTRN_MAG:* Make an edge magnitude image of the training image and put a pointer to it in the model data structure. An edge magnitude image is useful for tuning your model image so that it has only edges which characterize the pattern.
- *CGHT_RTRN_ANG:* Make an edge angle image of the training image and put a pointer to it in the model data structure. An angle image is useful for tuning your model image and for displaying your model.

Note: If you are training a model with **cght_train_seg()**, *flags is* ignored.

FUNCTION: **cght_transmit_model()**

PURPOSE: Converts a model from its internal format to a byte stream format for transfer to nonvolatile memory, to a serial or parallel port, or to other hardware. This function is independent of any hardware device. You select the device by passing a pointer to an output function.

REFERENCE: Vision Tools: Contour Finding Tool

INCLUDE FILE: *cght.h*

USE:

```
int cght_transmit_model(model,name,xfer_params
      output_func,output_params)
   cght_model *model;
   char *name;
   cio_xfer_params *xfer_params;
   c_function output_func;
   char *output_params;
```

NOTES: *model:* points to the model to transfer.

name: passed to the I/O function during initialization. This is usually used as a file name but may be NULL.

xfer_params: points to an initialized transfer parameters data structure. These parameters are used by all data structure transfer functions. This structure indicates the data compression style and the data transfer format.

You may specify NULL for this argument. **cght_transmit_model()** will create a temporary **cio_xfer_params** structure for you, with the following defaults:

style: packed
prep: ASCII
ascii: line feed terminator, 78 characters per line

output_func: points to a function to transmit the model to the hardware. You can write your own function or use one of the Cognex output functions: **cio_par_transmit()**, for sending data over parallel lines, or **cio_ser_transmit()**, for sending data over a serial line.

output_params: points to parameters to pass to the output function. If *output_func* is **cio_par_transmit()**, this should be a pointer to an initialized **cio_par_params** structure. If *output_func* is **cio_ser_transmit()**, this should be a pointer to an initialized **cio_ser_params** structure. If you are using your own output function, you can pass data to it with the *output_params* pointer.

cght_transmit_model() throws the following errors:

- *CGHT_ERR_BADARG,* if any argument is not valid
- *CGHT_ERR_PARAMS,* if a field of a parameters structure is not valid

FUNCTION: **cgm_close()**

PURPOSE: Perform a grey scale morphological closing on an image.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cip_buffer *cgm_close (source, dst, element, valid_dst)
   cip_buffer *source, *dst;
   cgm_element *element;
   cip_buffer *valid_dst;
```

NOTES: **cgm_close()** closes a source image using the specified structuring element and places the results in a destination image. If the source image is the same as the destination, then the result data replaces the source data.

Depending on the size and origin of the structuring element, some of the border pixels in the destination image will not be valid. The optional parameter *valid_dst* provides a mechanism for returning the valid region of the result image.

Note: This function operates on FAST8 images. If the source image is not FAST8 but is convertible, the function creates a FAST8 copy.

source points to the image to close.

dst points to the destination image. If *dst* is NULL, the function creates a destination image of the same size as the source image.

element points to a grey scale morphological structuring element.

valid_dst points to an image buffer that will hold the valid portion of the results. If *valid_dst* is non-NULL, **cgm_close()** returns the valid region of the result image. If NULL, *valid_dst* is ignored.

cgm_close() returns a pointer to the destination image.

cgm_close() throws the following errors:

- *CGEN_ERR_BADARG* if:
  - *dst* is specified and is not compatible with a FAST8 image, or is smaller than the source image in either the x or y dimension
  - *source* is NULL or invalid
  - *element* is NULL or invalid
  - you have specified an invalid origin for *element.origin*
- *CIP_ERR_NO_VC2* if a VC-2 coprocessor is not present

FUNCTION: **cgm_create_element()**

PURPOSE: Create a grey scale morphological structuring element.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cgm_element *cgm_create_element (radius, width, height,
      seed, element)
   int
      radius,
      width,
      height;
   short *seed;
   cgm_element *element;
```

NOTES: **cgm_create_element()** creates a grey scale morphological structuring element from user-specified parameters such as radius, or width and height. You may also specify 9 offset values to use as a seed in constructing these structuring elements; a non-flat seed will cause the resulting structuring element to be non-flat.

radius specifies the radius of a diamond-shape structuring element. If you specify a rectangular structuring element, *radius* must be set to zero.

width specifies the width of a rectangular structuring element. If you specify a diamond-shape structuring element, *width* must be set to zero.

height specifies the height of a rectangular structuring element. If you specify a diamond-shape structuring element, *height* must be set to zero.

seed points to an array of 9 **short** values to use as an initial 3x3 seed in constructing the structuring element. If NULL, a flat seed will be used.

element points to the destination **cgm_element** structure; if larger than 3x3, this is implemented as a linked list of 3x3 structuring elements which compose by dilation to a desired structuring element. If *element* is NULL, a destination **cgm_element** structure is allocated.

cgm_create_element() returns a pointer to the destination **cgm_element** structure.

cgm_create_element() throws *CGEN_ERR_BADARG* if:

- *radius* is zero and either *width* or *height* is zero
- *radius* is nonzero and either *width* or *height* is nonzero

FUNCTION: **cgm_create_element_view()**

PURPOSE: Convert a structuring element into an element view.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cgm_element_view *cgm_create_element_view (element,
      element_view)
   cgm_element *element;
   cgm_element_view *element_view;
```

NOTES: **cgm_create_element_view()** converts the specified structuring element into an element view.

element points to the structuring element to be converted.

element_view points to the destination element view structure. If *element_view* is NULL, **cgm_create_element_view()** allocates an element view. If *element_view* specifies an existing element view, **cgm_create_element_view()** deletes the existing structure and replaces it with the new one.

cgm_create_element_view() returns a pointer to the newly created **cgm_element_view** structure.

cgm_create_element_view() throws *CGEN_ERR_BADARG* if:

- the source structuring element is NULL or invalid
- you have specified an invalid origin, e.g. by ORing two valid origins, for *element.origin*

FUNCTION: **cgm_delete_element()**

PURPOSE: Delete a grey scale morphological structuring element.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
void *cgm_delete_element (element)
   cgm_element *element;
```

NOTES: **cgm_delete_element()** deletes the specified **cgm_element** structure using **free()**. If the **cgm_element** structure is a linked list, the entire list is deleted.

element points to the **cgm_element** structure to delete.

You can use **cgm_delete_element()** to delete **cgm_element** structures which have been created using **cgm_create_element()** or **cgm_make_lib()**. You cannot use **cgm_delete_element()** to delete **cgm_element** structures that have been allocated off the stack.

FUNCTION: **cgm_delete_element_view()**

PURPOSE: Delete an element view.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:
```
void *cgm_delete_element_view (element_view)
   cgm_element_view *element_view;
```

NOTES: **cgm_delete_element_view()** deletes the specified **cgm_element_view** structure. It calls **cip_delete()** on the **cip_buffer** components and **free()** on the element view structure.

element_view points to the **cgm_element_view** structure to be deleted.

FUNCTION: **cgm_dilate()**

PURPOSE: Perform a grey scale morphological dilation on an image.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cip_buffer *cgm_dilate (source, dst, element, valid_dst)
   cip_buffer *source, *dst;
   cgm_element *element;
   cip_buffer *valid_dst;
```

NOTES: **cgm_dilate()** dilates a source image using the specified structuring element and places the results in a destination image. If the source image is the same as the destination, then the result data replaces the source data.

Depending on the size and origin of the structuring element, some of the border pixels in the destination image will not be valid. The optional parameter *valid_dst* provides a mechanism for returning the valid region of the result image.

Note: This function operates on FAST8 images. If the source image is not FAST8 but is convertible, the function creates a FAST8 copy.

source points to the image to dilate.

dst points to the destination image. If *dst* is NULL, the function creates a destination image of the same size as the source image.

element points to a grey scale morphological structuring element.

valid_dst points to an image buffer that will hold the valid portion of the results. If *valid_dst* is non-NULL, **cgm_dilate()** returns the valid region of the result image. If NULL, *valid_dst* is ignored.

cgm_dilate() returns a pointer to the destination image.

cgm_dilate() throws the following errors:

- *CGEN_ERR_BADARG* if:
  - *dst* is specified and is not compatible with a FAST8 image, or is smaller than the source image in either the x or y dimension
  - *source* is NULL or invalid
  - *element* is NULL or invalid
  - you have specified an invalid origin for *element.origin*
- *CIP_ERR_NO_VC2* if a VC-2 coprocessor is not present

STRUCTURE: **cgm_element**

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

DEFINITION:

```
typedef struct cgm_element {
   short offsets[9];
   short origin;
   short dont_care_mask;
   struct cgm_element *next;
} cgm_element;
```

NOTES: The **cgm_element** structure contains the data for a grey scale morphology structuring element.

You can define a structuring element of up to 3x3 pixels by directly setting the components of a **cgm_element** structure (in this case *next* is set to NULL). To define structuring elements larger than 3x3, you must call **cgm_create_element()** or define a linked list of **cgm_element** structures.

offsets is a signed array of the 3x3 structuring element's offset values in the range -255 to 255. The offset values are specified in the order NW, N, NE, W, C, E, SW, S, SE, as shown below. Note that the structuring element is reflected prior to the morphological operation.

NW	N	NE
W	C	E
SW	S	SE

origin specifies the location of the structuring element's origin. *origin* must be one and only one of the following defines. Each define specifies a location within the 3x3 structuring element, as shown in *offsets*. Note that the structuring element is reflected prior to the morphological operation.

CIP_RANK_NE: upper right pixel

CIP_RANK_N: upper center pixel

CIP_RANK_NW: upper left pixel (default value of *origin*)

CIP_RANK_E: center right pixel

CIP_RANK_C: center pixel

CIP_RANK_W: center left pixel

CIP_RANK_SE: lower right pixel

CIP_RANK_S: lower center pixel

CIP_RANK_SW: lower left pixel

dont_care_mask specifies pixel locations within the structuring element to be ignored during the morphological operation. It can be one or more of the ORable defines listed for *origin*. Each define specifies a location within the 3x3 structuring element, as shown in *offsets*. If *dont_care_mask* is NULL, no mask is used.

next points to the next 3x3 mask in the linked list. NULL indicates that the structuring element is the last in the list.

STRUCTURE: **cgm_element_view**

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

DEFINITION:

```
typedef struct cgm_element_view {
   cip_buffer *offset_image;
   bool offset_image_signed;
   cip_buffer *mask_image;
   short origin_x, origin_y;
} cgm_element_view;
```

NOTES: The element view structure **cgm_element_view** is a viewable form of the grey scale morphology structuring element **cgm_element**.

offset_image points to an image of the structuring element's offset values. The offset image is always rectangular, regardless of the don't care mask specified (i.e. all offset values are shown, even if pixels are masked).

offset_image_signed specifies whether the offset image is a signed image. If TRUE, the offset image is signed. If FALSE (the default), the offset image is unsigned.

mask_image points to a binary image representing the structuring element's support. Pixels in the don't care mask are not shown in the mask image.

origin_x and *origin_y* specify the (x,y) coordinates of the structuring element's origin. The origin is always located within the offset image.

FUNCTION: **cgm_erode()**

PURPOSE: Perform a grey scale morphological erosion on an image.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cip_buffer *cgm_erode (source, dst, element, valid_dst)
   cip_buffer *source, *dst;
   cgm_element *element;
   cip_buffer *valid_dst;
```

NOTES: **cgm_erode()** erodes a source image using the specified structuring element and places the results in a destination image. If the source image is the same as the destination, then the result data replaces the source data.

Depending on the size and origin of the structuring element, some of the border pixels in the destination image will not be valid. The optional parameter *valid_dst* provides a mechanism for returning the valid region of the result image.

Note: This function operates on FAST8 images. If the source image is not FAST8 but is convertible, the function creates a FAST8 copy.

source points to the image to erode.

dst points to the destination image. If *dst* is NULL, the function creates a destination image of the same size as the source image.

element points to a grey scale morphological structuring element.

valid_dst points to an image buffer that will hold the valid portion of the results. If *valid_dst* is non-NULL, **cgm_erode()** returns the valid region of the result image. If NULL, *valid_dst* is ignored.

cgm_erode() returns a pointer to the destination image.

cgm_erode() throws the following errors:

- *CGEN_ERR_BADARG* if:
  - *dst* is specified and is not compatible with a FAST8 image, or is smaller than the source image in either the x or y dimension
  - *source* is NULL or invalid
  - *element* is NULL or invalid
  - you have specified an invalid origin for *element.origin*
- *CIP_ERR_NO_VC2* if a VC-2 coprocessor is not present

ENUMERATION: **cgm_lib_elements**

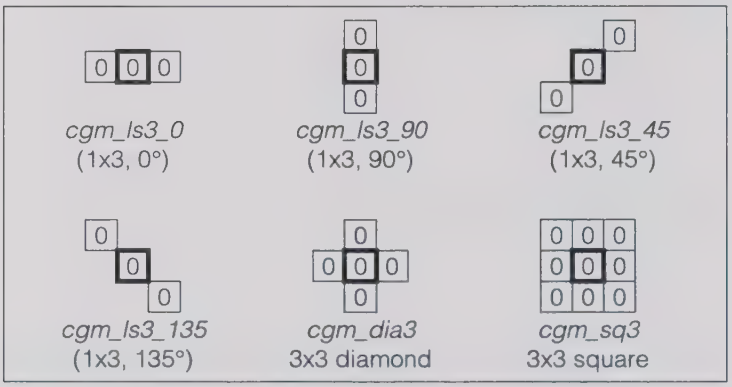
REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

DEFINITION:

```
typedef enum {
   cgm_ls3_0,
   cgm_ls3_90,
   cgm_ls3_45,
   cgm_ls3_135,
   cgm_dia3,
   cgm_sq3
} cgm_lib_elements;
```

NOTES: The **cgm_lib_elements** enumeration contains the structuring elements belonging to the Cognex grey scale morphological structuring element library. The structuring elements included in the Cognex library are illustrated below.



FUNCTION: **cgm_make_lib()**

PURPOSE: Convert a **cgm_lib_elements** enumeration constant into a **cgm_element** structure.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cgm_element *cgm_make_lib (lib_element, dst_element)
   cgm_lib_elements lib_element;
   cgm_element *dst_element;
```

NOTES: **cgm_make_lib()** converts the specified **cgm_lib_elements** enumeration constant into a **cgm_element** structure for use by the grey scale morphological operation functions. The grey scale morphological operation functions can take only structuring elements of type **cgm_element** (not **cgm_lib_elements**).

lib_element specifies the Cognex library structuring element to be converted to a **cgm_element** structure.

dst_element points to the converted structuring element. If *dst_element* is the name of an existing structuring element, **cgm_make_lib()** may delete it using **cgm_delete_element()** and reallocate it. If *dst_element* is NULL, **cgm_make_lib()** allocates a new structuring element.

Because **cgm_make_lib()** reserves the right to alter the structuring element that you pass it, you should call the function using one of the following conventions:

```
dst = cgm_make_lib(any_library_element, 0);
```

or

```
dst = cgm_make_lib(any_library_element, dst);
```

In contrast, you should not use the following calling convention:

```
cgm_make_lib(any_library_element, 0);
```

cgm_make_lib() returns a pointer to the converted structuring element.

FUNCTION: **cgm_open()**

PURPOSE: Perform a grey scale morphological opening on an image.

REFERENCE: Image Processing: Grey Scale Morphology

INCLUDE FILE: *cgm.h*

USE:

```
cip_buffer *cgm_open (source, dst, element, valid_dst)
   cip_buffer *source, *dst;
   cgm_element *element;
   cip_buffer *valid_dst;
```

NOTES: **cgm_open()** opens a source image using the specified structuring element and places the results in a destination image. If the source image is the same as the destination, then the result data replaces the source data.

Depending on the size and origin of the structuring element, some of the border pixels in the destination image will not be valid. The optional parameter *valid_dst* provides a mechanism for returning the valid region of the result image.

Note: This function operates on FAST8 images. If the source image is not FAST8 but is convertible, the function creates a FAST8 copy.

source points to the image to open.

dst points to the destination image. If *dst* is NULL, the function creates a destination image of the same size as the source image.

element points to a grey scale morphological structuring element.

valid_dst points to an image buffer that will hold the valid portion of the results. If *valid_dst* is non-NULL, **cgm_open()** returns the valid region of the result image. If NULL, *valid_dst* is ignored.

cgm_open() returns a pointer to the destination image.

cgm_open() throws the following errors:

- *CGEN_ERR_BADARG* if:
  - *dst* is specified and is not compatible with a FAST8 image, or is smaller than the source image in either the x or y dimension
  - *source* is NULL or invalid
  - *element* is NULL or invalid
  - you have specified an invalid origin for *element.origin*
- *CIP_ERR_NO_VC2* if a VC-2 coprocessor is not present

STRUCTURE: **cgr_caliper**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_cal.h*

DEFINITION:

```
typedef struct {
   cgr_figure h;
   cgr_caliper_params p;
   cgr_caliper_contour c;
   cia_point ap;
}  cgr_caliper;
```

NOTES: **cgr_caliper** defines the characteristics of a caliper figure.

h: the general figure data.

p: an instance of the **cgr_caliper_params** structure which defines various parameters for the caliper figure.

c: an instance of the **cgr_caliper_contour** structure which defines the points used for drawing the caliper figure. These points are automatically derived.

ap: the derived anchor point for the figure. This anchor point is automatically derived from information specified in other structure components.

FUNCTION: **cgr_caliper_init()**

PURPOSE: Initializes a caliper figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_cal.h*

USE:

```
cgr_figure *cgr_caliper_init(fig,p,h)
    cgr_caliper *fig;
    cgr_caliper_params *p;
    cgr_figure *h;
```

NOTES: *fig*: points to a caliper figure. If *fig* is null, a caliper figure is allocated and a pointer to it is returned.

p: points to the caliper figure parameters. If *p* is null, the figure parameters are set to the following values:

element	value
ap	0,0
mr	50,10
p1	10
p2	10
b	0

h: points to the general figure data structure. If *h* is null, a data structure is created with the following values:

element	value
draw	cgr_caliper_draw
derive	cgr_caliper_derive
update	cgr_caliper_update
lbut	cif_std_lbut
mbut	cif_std_mbut
rbut	cif_std_rbut
train	0

element	value
total_size	size_of(cgr_caliper)
nm	5

STRUCTURE: **cgr_caliper_params**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_cal.h*

DEFINITION:

```
typedef struct {
   cia_point
      ap,
      mr;
   int p1, p2;
   unsigned short b;
}  cgr_caliper_params;
```

NOTES: **cgr_caliper_params** indicates a caliper's anchor point, size, position of edges, and orientation. These parameters must be supplied; they are not automatically derived.

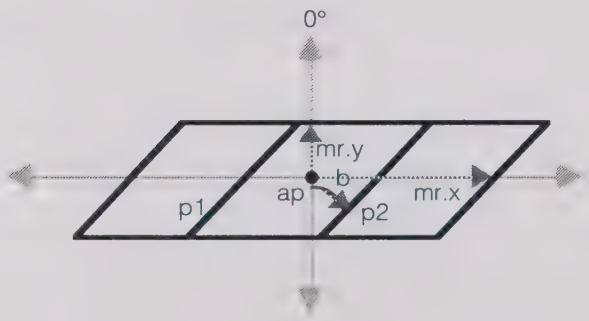
ap: the anchor point of the caliper, specified in physical units.

mr: the main contour radii of the caliper figure, specified in physical units.

p1, p2: the position of the inner edges of the caliper. *p1* and *p2* are specified in physical units relative to the center of the caliper. *p1* is the leftmost inner edge, and *p2* is the rightmost inner edge.

b: the orientation of the caliper, specified in binary.

The following diagram shows the relationship of **cgr_caliper_params** elements to a caliper figure.



FUNCTION: **cgr_character()**

PURPOSE: Draws a character into an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_character(image,x,y,code,ink,background)
   cip_buffer *image;
   int x, y;
   char code,ink,background;
```

NOTES: The character is a 5 x 9 dot matrix character with descending lower case. Each character is drawn in an 8 x 10 block of pixels.

image: points to the image into which to write the character.

x and *y:* the image coordinates at which the upper left corner of the character is written.

code: the ASCII code for the character. Characters available are the ASCII characters from space (040) to delete (0177).

ink: the color of the character.

background: the color of the character's background. The background is that part of the 8 x 10 block that is not occupied by the character itself. Specifying -1 leaves the background untouched, drawing only *ink* in the image.

cgr_character() is most useful for drawing characters at arbitrary locations in an image. You must calculate the position where each character will be displayed. Characters need not line up horizontally or vertically or be equally spaced; e.g., half spacing such as subscripts or superscripts is allowed. If you want to write strings of characters without calculating the individual character positions, you should use **cgr_string()** or **cgr_printf()**.

cgr_character() cannot be used with *CGR_PUSH* draw mode. Use the **cgr_string()** function to write characters when using *CGR_PUSH* draw mode.

cgr_character() throws the following errors:

- *CGEN_ERR_BADARG* if you try to write a character that is not within the ASCII range 040 (space) to 0177 (delete)
- *CIP_ERR_PELADDR* if you try to write a character outside of the image
- *CIP_ERR_PELDEPTH* if you attempt to write a character with too large an *ink* value

FUNCTION: **cgr_circle()**

PURPOSE: Draws a circle of a specified grey level in an image and returns a pointer to the image. **cgr_circle()** is provided for compatibility with previous software releases. It is recommended that you use **cgr_ellipse()** to draw circles

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_circle(image,x0,y0,radius,gcw)
   cip_buffer *image;
   int x0, y0, radius;
   int gcw;
```

NOTES: *image:* points to the image in which to draw the circle.

x0, y0: the x and y coordinates of the center of the circle.

radius: the radius of the circle.

gcw: the graphics control word.

cgr_circle() throws the following errors:

- *CNOTIMP_ERR* if the image is not *FAST8*
- *CIP_ERR_PELADDR* if the circle does not fit within image

STRUCTURE: **cgr_closed**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *contour.h*

DEFINITION:

```
typedef struct {
   cgr_figure h;
   cgr_synthetic_params p;
   cgr_closed_contour
      main,
      outer,
      inner;
   cia_point ap;
   short sn, cs;
   int
      speeds[9],
      min_thick;
}  cgr_closed;
```

NOTES: **cgr_closed** defines the characteristics of a synthetic rectangle model figure.

h: the general figure data.

p: an instance of the **cgr_synthetic_params** structure which defines various parameters for synthetic models.

main: an instance of the **cgr_closed_contour** structure for defining the geometric points of the main corner. These points are automatically derived.

outer: an instance of the **cgr_closed_contour** structure for defining the geometric points of the inner corner of the figure. These points are automatically derived.

inner: an instance of the **cgr_closed_contour** structure for defining the geometric points of the inner corner of the figure. These points are automatically derived.

ap: the derived anchor point for the figure.

sn, cs: the derived sine and cosine of the orientation angle.

speeds: an array of type **int** which lists the trackball update speeds for the figure. The update speed indicates the number of physical units that the figure is moved for every trackball tick. The *speeds* array maps trackball speeds with a figure's update modes.

min_thick: the minimum number of pixels that must appear between contours in the figure.

FUNCTION: **cgr_closed_init()**

PURPOSE: Initializes a synthetic rectangle model figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *contour.h*

USE:

```
cgr_figure *cgr_closed_init(fig,p,h)
   cgr_closed *fig;
   cgr_synthetic_params *p;
   cgr_figure *h;
```

NOTES: *fig*: points to a synthetic rectangle model figure. If *fig* is null, a synthetic rectangle model figure is allocated and a pointer to it is returned.

p: points to the synthetic rectangle model figure parameters. If *p* is null, the figure parameters are set to the following values:

element	value
ap	0,0
mr	32*SQRT2, 32*SQRT2
or	32<<16, 32<<16
ir	0,0
mc	0,0
oc	0,0
ic	0,0
b	0

h: points to the general figure data structure. If *h* is null, a data structure is created with the following values:

element	value
draw	cgr_synthetic_closed_draw
derive	cgr_synthetic_closed_derive
update	cgr_synthetic_closed_update
lbut	cif_std_lbut
mbut	cif_std_mbut

element	value
rbut	cif_std_rbut
train	NULL
total_size	size_of(cgr_closed)
nm	9
z	CAQ_WHITE
zp	CAQ_BLINK
zoff	CAQ_PASS
x0	caq_image->width/2
y0	caq_image->height/2
xres	full res = 0x200 (2), half res = 0x100 (1)
yres	full res = 0x200 (2), half res = 0x100 (1)

The following default update speed settings are used if none are provided:

array element	corresponding update mode	speed (physical units/trackball tick)
default_closed_speeds[0]	coarse position	0x20000 = 2.0
default_closed_speeds[1]	orientation	0x80 = 0.703°
default_closed_speeds[2]	fine position	0x4000 = 0.25
default_closed_speeds[3]	main rect. size	0x4000 = 0.25
default_closed_speeds[4]	main rect. corner	0x8000 = 0.5
default_closed_speeds[5]	outer rect. size	0x4000 = 0.25
default_closed_speeds[6]	outer rect. corner	0x8000 = 0.5
default_closed_speeds[7]	inner rect. size	0x4000 = 0.25
default_closed_speeds[8]	inner rect. corner	0x8000 = 0.5

STRUCTURE: **cgr_corner**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *contour.h*

DEFINITION:

```
typedef struct {
   cgr_figure h;
   cgr_synthetic_params p;
   cgr_corner_contour
      main,
      outer,
      inner;
      cia_point ap;
   short sn, cs;
   int
      speeds[9],
      min_thick;
}  cgr_corner;
```

NOTES: **cgr_corner** defines the characteristics of a synthetic corner model figure.

h: the general figure data.

p: an instance of the **cgr_synthetic_params** structure which defines various parameters for synthetic models.

main: an instance of the **cgr_corner_contour** structure for defining the geometric points of the corner. These points are automatically derived.

outer: an instance of the **cgr_corner_contour** structure for defining the geometric points of the inner corner of the figure. These points are automatically derived.

inner: an instance of the **cgr_corner_contour** structure for defining the geometric points of the inner corner of the figure. These points are automatically derived.

ap: the derived anchor point for the figure. This anchor point is automatically derived from information specified in other structure components.

sn, cs: the derived sine and cosine of the orientation angle.

speeds: an array of type **int** which lists the update speeds for the trackball. The update speed indicates the number of physical units that the figure is moved for every trackball tick. The *speeds* array maps trackball speeds with a figure's update modes.

min_thick: the minimum number of pixels that must appear between contours in the figure.

FUNCTION: **cgr_corner_init()**

PURPOSE: Initializes a synthetic corner model figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *contour.h*

USE:

```
cgr_figure *cgr_corner_init(fig,p,h)
   cgr_corner *fig;
   cgr_synthetic_params *p;
   cgr_figure *h;
```

NOTES: *fig*: points to a synthetic corner model figure. If *fig* is null, a synthetic corner model figure is allocated and a pointer to it is returned.

p points to the synthetic corner model figure parameters. If *p* is null, the figure parameters are set to the following values:

element	value
ap	0,0
mr	64SQRT2, 64*SQRT2
or	64<<16, 64<<16
ir	0,0
mc	0,0
oc	0,0
ic	0,0
b	0

h: points to the general figure data structure. If *h* is null, a data structure is created with the following values:

element	value
draw	cgr_synthetic_corner_draw
derive	cgr_synthetic_corner_derive
update	cgr_synthetic_corner_update
lbut	cif_std_lbut
mbut	cif_std_mbut

element	value
rbut	cif_std_rbut
train	NULL
total_size	size_of(cgr_corner)
nm	9
z	CAQ_WHITE
zp	CAQ_BLINK
zoff	CAQ_PASS
x0	caq_image->width/2
y0	caq_image->height/2
xres	full res = 0x200 (2), half res = 0x100 (1)
yres	full res = 0x200 (2), half res = 0x100 (1)

The following default update speed settings are used if none are provided:

array element	corresponding update mode	speed (physical units/trackball tick)
default_corner_speeds[0]	coarse position	0x20000 = 2.0
default_corner_speeds[1]	orientation	0x80 = 0.703°
default_corner_speeds[2]	fine position	0x4000 = 0.25
default_corner_speeds[3]	main corner size	0x4000 = 0.25
default_corner_speeds[4]	main corner	0x8000 = 0.5
default_corner_speeds[5]	outer corner size	0x4000 = 0.25
default_corner_speeds[6]	outer corner	0x8000 = 0.5
default_corner_speeds[7]	inner corner size	0x4000 = 0.25
default_corner_speeds[8]	inner corner	0x8000 = 0.5

FUNCTION: **cgr_cross()**

PURPOSE: Displays a cross of the specified dimensions and pixel value at the specified location in an image, and returns a pointer to the image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_cross(image, x0, y0, x_arm, y_arm, gcw)
   cip_buffer *image;
   int x0, y0, x_arm, y_arm, gcw;
```

NOTES: *image:* points to the image in which the cross is drawn.

x0, y0: the location at which the arms of the cross intersect.

x_arm, y_arm: the distances that the horizontal and vertical arms of the cross extend from *(x0,y0)*. If an arm of the cross extends beyond the edge of the image, the arm is truncated. *x_arm* and *y_arm* can be any nonnegative integer. If you specify 0 for either *x_arm* or *y_arm*, that arm is not drawn; if both arguments are 0, a point is drawn at *(x0,y0)*.

gcw: the graphics control word.

cgr_cross() throws the following errors:

- *CIP_ERR_PELADDR* if *x0* and *y0* are not within the image, or if there is only one row in the image
- *CGEN_ERR_BADARG* if either *x_arm* or *y_arm* is negative
- *CIP_ERR_PELDEPTH* if the value specified by the GCW is outside of *image*'s pixel depth

FUNCTION: **cgr_display_image()**

PURPOSE: Displays a magnified image within another image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:
```
cip_buffer *cgr_display_image(src,dest,x0,y0,mag)
   cip_buffer *src, *dest;
   int x0, y0, mag;
```

NOTES: **cgr_display_image()** returns a pointer to the destination image.

src: points to the source image.

dest: points to the destination image.

Since **cgr_display_image()** calls **cip_window()** for use on both the source image and the destination image, both of these images must be images in which you can create a window. The frame grabber, any window of the frame grabber, or any image created by a **cip_** function, meets this requirement.

x0, *y0:* the location in the destination image at which the origin (0,0) of the source image is placed. If either *x0* or *y0* is -1, the source image is centered in that dimension of the destination image.

mag: specifies how much to magnify the source image. *mag* must be a positive number less than or equal to 32,767. Each pixel of the source image becomes a *block* of *mag by mag* pixels in the destination image.

cgr_display_image() does not reconcile the pixel sizes of the source and destination images. Before calling this function, you may need to apply a function such as **cip_pixel_map()** to the source image to adjust the pixel value range in order to get visually satisfactory or legal results.

For example, if the source image pixel depth is less than the destination image pixel depth, the destination image will not contain the full range of available pixel values and so may have low contrast. In this case, you can use **cip_pixel_map()** to map the range of grey values in the source image into a wider range of grey values in the destination image.

If the source image cannot fit within the destination image at the specified location and magnification, the source image is truncated to fit, and no error is reported. If the magnification of a row or column of pixels in the truncated image is partly inside and partly outside of the destination image, that row or column may be omitted, resulting in the image not extending to the edge the destination image.

The source image and the destination image may share some or all of their pixels when **cgr_display_image()** is called. This may happen if the source and destination are either the same image or windows of the same image. In

this case, the source image is first copied to a temporary space before it is copied into the destination image.

You can use **cgr_display_image()** to see if an image digitized the way you expected. You also can use this function to create custom control screens; for example, a screen containing a display showing the different models available for use on a flexible assembly line where the tasks, and therefore the models, may change from day to day.

cgr_display_image() is implemented using **cip_expand()**. One advantage to using **cgr_display_image()** instead of just **cip_expand()** is that **cgr_display_image()** does special handling when the source and destination images overlap. **cgr_display_image()** also allows you to specify the location at which you want the image magnified; **cip_expand()** does not.

cgr_display_image() throws the following errors:

- *CGEN_ERR_BADARG* if the magnification is zero or negative, or if it is too large to fit in a **short**
- *CIP_ERR_PELDEPTH* if the source image depth is greater than the destination image depth (meaning that some source pixels will not fit in the destination image unless they are scaled) and the source image is not scaled; or if a pixel to be copied from the source image is too big to fit in the destination image's pixel size
- *CVC_ERR_ALLOC* if temporary space is limited and the source and destination are either the same image or windows of the same image, causing the source image to be copied to temporary space before it is copied into the destination image

FUNCTION: **cgr_display_model()**

PURPOSE: Displays a magnified model within an image and puts a cross at the model origin.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_display_model(model,image,x0,y0,mag,arm,
      color)
   cse_model *model;
   cip_buffer *image;
   int x0, y0, mag, arm, color;
```

NOTES: Typically, you use **cgr_display_model()** to see what the image of a model looks like. For example, you may want to see if a model has digitized as you expected. This is easier to do with a binary model than with a grey model, since it may be difficult to see slight differences in the grey values of a grey model. **cgr_display_model()** has applications similar to those for **cgr_display_image()**. **cgr_display_model()** only generates output grey values in the pixel range 0 to 63.

model: points to the model to be magnified and displayed.

image: points to the destination image, which must be an image in which you can create a window. The frame grabber, any window of the frame grabber, or any image created by a **cip_** function, is this type of image.

x0, y0: the image coordinates at which the upper left corner (0,0) of the model is placed. If *x0* is -1 then the model is centered horizontally in the image. If *y0* is -1 then the model is centered vertically in the image.

mag: the model magnification. This should be a positive number, small enough to fit in a **short**. Each pixel in the model becomes a *block* of *mag by mag* pixels in the image.

arm: the length of each of the four arms that make up the cross drawn at the model origin by **cgr_display_model()**.

The cross is drawn even if the model's origin is outside the portion of the image where the model is displayed. If an arm of the cross extends outside the image, the arm is truncated. If the model's origin is outside the image, then the cross is not drawn and no error is reported.

color: the color of the cross. It may have values 0-63.

The global variable *cgr_mod_dontcare* contains the value to use as the representation for a "don't care" pixel in the model. Values close to this value are not used to display ordinary
model pixels.

cgr_mod_dontcare can be set at any time, but we recommend that you set it only at boot time since its value usually remains unchanged for a given environment. The default value of 0 is satisfactory for grey models.

cgr_display_model() passes the image component of the model to **cgr_display_image()** which does the bulk of the processing for **cgr_display_model()**. Therefore, the usage and restrictions of **cgr_display_image()** also apply to **cgr_display_model()**.

A model contains an image, but it also contains model data such as the model origin, resolution and number of grey levels. **cgr_display_model()** displays only the visual information about a model. It does not display any data about the model; you must use **cd_show()** to see model data.

If the image cannot contain the model at the specified location and magnification, the model is truncated to fit according to the style of **cgr_display_image()** and no error is reported.

cgr_display_model() does not check for any errors associated with its arguments, since all necessary checking is done when it calls **cgr_display_image()**. See the description of **cgr_display_image()** for the errors that may be thrown as a result of this checking.

cgr_display_image() throws *CGR_ERR_DONTCARE* if the value of *cgr_mod_dontcare* is out of range for the pixel depth of the image.

FUNCTION: **cgr_ellipse()**

PURPOSE: Draws an ellipse arc in an image and returns a pointer to the image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
*cip_buffer *cgr_ellipse(image,x0,y0,rx,ry,theta,
      phi1,phi2,da,gcw)
   cip_buffer *image;
   int
      x0, y0,
      rx, ry;
   unsigned short
      theta,
      phi1, phi2;
   int da, gcw;
```

NOTES: The ellipse may be partially or totally outside the image. The function draws only that portion of the ellipse that is inside the image.

image: the image in which to display the ellipse.

x0, y0: the x and y coordinates, in pixels, of the center of the ellipse. *x0* and *y0* are of type **CIA_31_16**.

rx, ry: the x and y radii of the ellipse, in pixels. *rx* and *ry* are of type **CIA_31_16**. The maximum radius allowable is 2047.9375. The minimum radius allowable is 2.

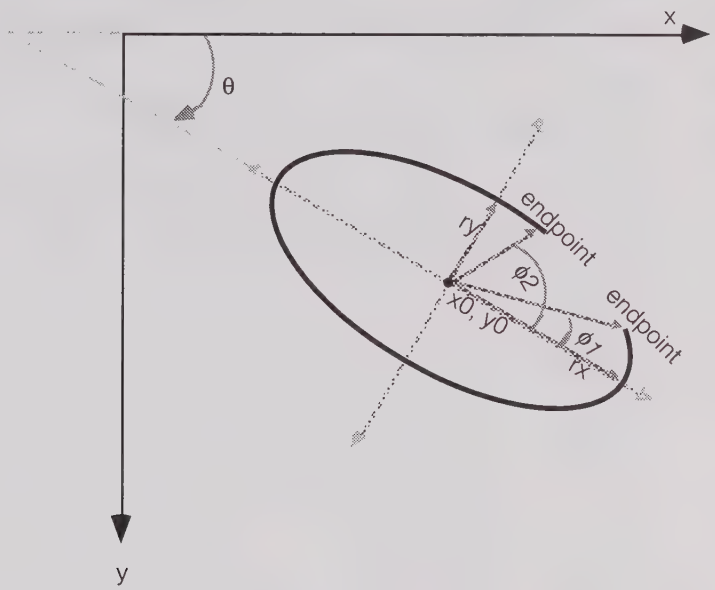
theta: the angle of orientation for the ellipse. *theta* is a binary angle specification of type **CIA_31_16** of the form 16.16.

phi1, phi2: endpoint specifications for the ellipse arc. If phi1 and phi2 are equal, the ellipse arc is a closed ellipse. *phi1* and *phi2* are binary angle specifications of type **CIA_31_16** of the form 16.16.

da: reserved for future use.

gcw: the graphics control word.

The following diagram shows the relationship of **cgr_ellipse()** arguments to an ellipse arc.



cgr_ellipse() throws the following errors:

- *CNOTIMP_ERR* if the draw bit value of the GCW is invalid

VARIABLE: **cgr_fgscreen**

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

DEFINITION:

```
cgr_screen *cgr_fgscreen;
```

NOTES: Points to an instance of the structure, **cgr_screen,** for use as an argument to the function **cgr_printf()** for writing formatted output to the monitor. The values of the elements of *cgr_fgscreen* are as follows:

```
**ibuff      &caq_image,
*font        &std_font
cellwd       8
cellht       22
cursorx      0
cursory      0
flags        cgr_sc_newline | cgr_sc_wrap
ink          CAQ_BLACK
backgr       CAQ_WHITE
scink        CAQ_BLACK
scbackgr     CAQ_WHITE
cursorink    CAQ_BLINK
escargs      0
escarg[2]    0,0
```

For all practical purposes, the values of the following elements are fixed: *font=&std_font, cellwd=8, cellht=22.* The font is fixed because there are no tools for creating screen fonts. Cell width and height can be made larger; however, it is not practical to do so because the height and width of the characters in the font are fixed.

The following elements are manipulated by **cgr_printf()** and you are advised not to change them: *flags, escargs, escargs[0], escargs[1].*

You may change the values of the following elements directly: *cursorx, cursory, ink, backgr, scink, scbackgr, cursorink.*

The default ink and background values are for use with overlay graphics; i.e., a call to **caq_graphics()** after the call to **cgr_printf()**. To write on a frozen image, change the values of the ink and background to non-graphic values and follow the call to **cgr_printf()** with a call to **caq_display()**.

MACRO: **CGR_FIG_IMG()**

PURPOSE: Convert figure physical unit measurements (**CIA_31_16**) to image coordinates (**CIA_31_16**).

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

USE:

```
#define CGR_FIG_IMGX(fig, x) (((x) >> 8) * (fig)->xres +
                              ((fig)->x0 << 16))
#define CGR_FIG_IMGY(fig, y) (((y) >> 8) * (fig)->yres +
                              ((fig)->y0 << 16))
```

NOTES: *fig* is the figure to which the physical unit measurements apply

x: the physical unit specification in x.

y: the physical unit specification in y.

STRUCTURE: **cgr_figure**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

DEFINITION:

```
typedef struct {
   short
      xmin, ymin,
      wd, ht,
      xres, yres,
      x0, y0;
   int total_size;
   c_function
      draw,
      derive,
      update,
      lbut,
      mbut,
      rbut;
   cse_model *(*train)();
   char
      mode,
      nm;
   int
      z,
      zp,
      zoff;
} cgr_figure;
```

NOTES: **cgr_figure** contains information common to all graphics figures.

xmin, *ymin*, *wd*, *hd*: define the enclosing rectangle for the figure. These structure components are derived automatically based on figure parameters.

xres, *yres*: indicate the number of pixels per physical unit and are of type **CIA_15_8**. For example, if 0x100 is specified for *xres* and 0x200 is specified for *yres*, the physical unit specification 4,5 would be interpreted as 4,10 in pixel units.

x0, *y0*: the location of physical (0,0) on the screen. *x0* and *y0* are pixel values. For example, on a 512 x 512 screen, specifying 256 for both *x0* and *y0* places physical (0,0) in the middle of the screen.

total_size: the total size, in bytes, of the figure structure for the figure being defined. The total size is based on the general figure data and all figure-specific data.

draw: the C function used to draw the figure in the image.

derive: the C function that is typically used to calculate derived figure values and determines if the proposed drawing of a figure will fall within the

boundaries of the image. However, the derive function can be used for any purpose.

update: the C function that updates a figure according to the current update mode.

lbut: the C function that performs an action when the left trackball button is pressed.

mbut: the C function that performs an action when the middle trackball button is pressed.

rbut: the C function that performs an action when the right trackball button is pressed.

train: points to a training function, if the figure is to be trained as a model.

mode: the current update mode.

nm: the number of update modes for the figure.

z: the graphics control word for the figure.

zp: the graphics control word for the figure pointer.

zoff: the background color. This component is relevant only if the figure is drawn with the draw bit of the GCW set to *CGR_SET*.

FUNCTION: **cgr_fill()**

PURPOSE: Fills all pixels of a specific grey value, in a specified region of an image, with a new grey value.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_fill(image,x,y,fill_gl,back_gl)
   cip_buffer *image;
   int x,y, fill_gl, back_gl;
```

NOTES: This function can be used only on *regular* images. Therefore, it cannot be used on 2000 frame grabber images or 4400 frame grabber images that are not in 512 x 512 mode.

This function does a *4-connected fill;* that is, it fills the four pixels connected to each pixel of the specified grey value with the new grey value. Pixels diagonally adjacent to the pixel are not considered connected and are not filled. If a connected pixel is not the specified grey value, that pixel is not filled with the new grey value.

The region being filled must be *bounded* by a string of connected pixels with grey values different from the grey value that is being replaced. If you cannot be sure that the image will be bounded, create an image buffer larger than you need and surround the image with a grey level other than the one being replaced, then call **cgr_fill()** on a window of the image buffer.

image: points to the image in which to perform the fill. The fill must be done in a bounded region of the image.

x, y: the (x,y) coordinates of the point at which to begin filling.

fill_gl: the grey level with which to fill the pixels.

back_gl: the grey level of the pixels to be replaced by the fill. If the grey value of the pixel at (x,y) is not *back_gl*, no fill is performed.

cgr_fill() throws *CGEN_ERR_BADARG* if *image* is not regular.

FUNCTION: **cgr_grid()**

PURPOSE: Draws a "graph paper" background on an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_grid(image,nx,xcode,ny,ycode,gcw)
   cip_buffer *image;
   int nx, xcode, ny, ycode, gcw;
```

NOTES: **cgr_grid()** can draw the grid to linear or square root scales.

image: points to the image in which to draw the grid background.

nx: the number of parts into which to divide the image along the x axis. For example, if you specify 10, the image is divided into 10 parts from left to right.

xcode: specifies whether the scale of the horizontal interval is linear or square root. This determines whether the image is divided into equal sized parts (linear scale), or whether the parts decrease in size from left to right (square root scale).

You specify scale type with the constants *CGR_GC_LINEAR* and *CGR_GC_SQRT*.

On a square root graph, the lines are actually drawn at the square root of the value displayed. Square root scale allows for a larger portion of the display to be dedicated to the lower end of the scale (0-30 percent) which is particularly useful when your data consists largely of small numbers. For example, if you have a graph from 0 to 100 where a line is drawn at every tenth value, the line is actually drawn at the square root of that value.

ny: same as *nx*, above, but for the y axis.

ycode: same as *xcode*, above, but for the vertical interval.

gcw: the graphics control word.

STRUCTURE: **cgr_gwindow**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gwindow.h*

DEFINITION:

```
typedef struct {
   cgr_figure h;
   cgr_gwindow_params p;
   cgr_gwindow_points c;
   cia_point ap;
   short sn, cs;
   int speeds[5];
}  cgr_gwindow;
```

NOTES: **cgr_gwindow** defines the characteristics of a generalized window figure.

h: the general figure data.

p: an instance of the **cgr_gwindow_params** structure which defines various parameters for the window.

c: an instance of the **cgr_gwindow_points** structure which defines the points used for drawing the window figure. These points are automatically derived.

ap: the derived anchor point, in pixels, for the figure. This anchor point is automatically derived from information specified in other structure components.

sn, cs: the derived sine and cosine of the orientation angle.

speeds: an array of type **int** which lists the trackball update speeds for the figure. The update speed indicates the number of physical units that the figure is moved for every trackball tick. The *speeds* array maps a figure's update modes to trackball speeds.

FUNCTION: **cgr_gwindow_init()**

PURPOSE: Initializes a generalized window figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gwindow.h*

USE:

```
cgr_figure *cgr_gwindow_init(fig,p,h)
   cgr_gwindow *fig;
   cgr_gwindow_params *p;
   cgr_figure *h;
```

NOTES: *fig*: points to a window figure. If *fig* is null, a window figure is allocated and a pointer to it is returned.

p: points to the window figure parameters. If *p* is null, the window figure parameters are set to the following values:

element	value
ap	0,0
mr	20<<16, 20<<16
mc	0,0
b	0

h: points to the general figure data structure. If *h* is null, a data structure is created with the following values:

element	value
draw	cgr_gwindow_draw
derive	cgr_gwindow_derive
update	cgr_gwindow_update
lbut	cif_std_lbut
mbut	cif_std_mbut
rbut	cif_std_rbut
train	0
total_size	size_of(cgr_gwindow)
nm	5

element	value
z	CAQ_WHITE
zp	CAQ_BLINK
zoff	CAQ_PASS
x0	caq_image->width/2
y0	caq_image->height/2
xres	full res = 0x200 (2), half res = 0x100 (1)
yres	full res = 0x200 (2), half res = 0x100 (1)

The following default update speed settings are used if none are provided:

array element	corresponding update mode	speed (physical units/trackball tick)
default_gwindow_speeds[0]	coarse position	0x20000 = 2.0
default_gwindow_speeds[1]	orientation	0x80 = 0.703°
default_gwindow_speeds[2]	fine position	0x4000 = 0.25
default_gwindow_speeds[3]	main radii	0x4000 = 0.25
default_gwindow_speeds[4]	corner radii	0x8000 = 0.5

STRUCTURE: **cgr_gwindow_params**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gwindow.h*

DEFINITION:

```
typedef struct {
   cia_point ap, mr, mc;
   unsigned short b;
}  cgr_gwindow_params;
```

cgr_gwindow_params indicates a window's anchor point, the curvature of the window corners, and the angle of rotation of the window. These parameters must be supplied; they are not automatically derived.

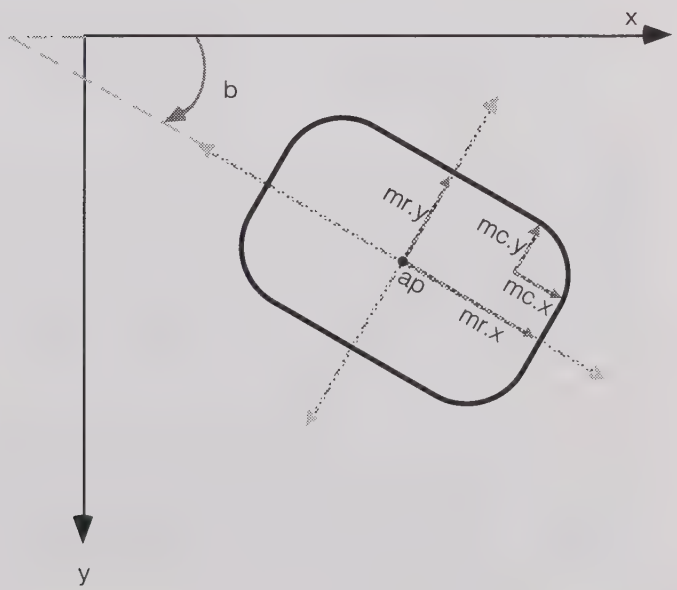
ap: the anchor point of the window, specified in physical units. The anchor point is the point about which the window is rotated.

mr: the main contour radii of the window, specified in physical units.

mc: the main corner radii of the window, specified in physical units.

b: the angle of rotation of the window, specified in binary. Rotation is clockwise.

The following diagram shows the relationship of **cgr_gwindow_params** elements to a window figure.



FUNCTION: **cgr_histogram()**

PURPOSE: Displays a histogram of an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_histogram(image,hist,size)
   cip_buffer *image;
   int *hist,size;
```

NOTES: A histogram is a one-dimensional function of image grey value that represents the distribution of grey values in an image. **cgr_histogram()** displays both a histogram and its *cumulative*. The cumulative is a representation of the integral of the histogram, where a point on the cumulative represents the percentage of all pixel grey values less than or equal to the grey value at that point. Histogram displays can aid in determining the best values for mapping parameters and thresholds. They can also be used to estimate the area associated with specific grey values. **cgr_histogram()** returns a pointer to the image in which the histogram is drawn.

image: points to the image into which to write the histogram display.

hist: points to the array of data to be displayed; that is, the histogram. The function **cip_zhist()** computes and returns both an intensity histogram of an image and a pointer to the bins of the histogram. Specify this pointer as the *hist* argument.

size: the number of bins in *hist* to display. A histogram is represented as an array of integers where each element (also called a *bin*) of the array holds a count of the number of pixels in the image of the grey value that equals the array index of the bin.

cgr_histogram() writes a histogram into an image in bar graph form. The y axis of the bar graph is divided into 10 parts, with each part representing 10 percent of the total number of elements comprising the histogram. The x axis is divided into eight parts, with each part representing 1/8 of the total number of grey levels (bins). Since a majority of the bin values will be less than 10 percent of the total amount of histogram elements, a square root compressed scale in the y dimension is used for the display. This scale allows for a larger portion of the display to be dedicated to the lower end of the scale (0-30 percent) where, typically, most of the bin values fall.

The cumulative display is useful for determining values for various mapping parameters. For example, if you wanted to perform relative linear mapping, you could use the cumulative display to estimate values for the components of the data structure **re** of the **cmap_map_data** union: *re->left_p*, *re->right_p*, *re->zero*, and *re->one*. The cumulative display could also be used to select a binary threshold for binary mapping.

cgr_histogram() throws *CGEN_ERR_BADARG* if the histogram is wider than the **cip_buffer** in which it is to be displayed.

FUNCTION: **cgr_iline()**

PURPOSE: Identical to **cgr_line()**, except that **cgr_iline()** will clip a line if it exceeds the boundary of an image, rather than throw a *CIP_ERR_PELADDR* error.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_iline(image,x0,y0,x1,y1,gcw)
   cip_buffer *image;
   int x0, y0, x1, y1, gcw;
```

NOTES: If a line exceeds the boundary of an image, **cgr_line()** uses floating point calculations to clip the line at the image boundary. If the system does not include an FPU, the drawing process may be slowed when a line needs to be clipped.

image: points to the image in which the line is drawn.

x0, *y0*, *x1*, *y1:* the end points of the line and must lie within the image.

gcw: the graphics control word.

cgr_iline() throws *CIP_ERR_PELDEPTH* if the color value specified in the graphics control word (GCW) cannot fit in the image's pixel size.

FUNCTION: **cgr_image()**

PURPOSE: Draws an image in another image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *gstack.h*

USE:

```
void cgr_image(source,dest,x,y,gcw)
   cip_buffer *source, *dest;
   int x, y, gcw;
```

NOTES: *source*: points to the image to be drawn.

dest: points to the image in which to display the image you drawing. Typically, *dest* is *caq_image*.

x, y: the coordinates in the destination image at which to place the upper left corner of the source image.

gcw: the graphics control word.

cgr_image() throws the following errors:

- *CGR_ERR_ALLOC* if there is not enough room on the graphics stack for the image graphics frame
- *CNOTIMP_ERR* if you specify a draw setting other than *CGR_SET* or *CGR_PUSH* for the GCW

FUNCTION: **cgr_line()**

PURPOSE: Draws a line segment of a specified pixel value between two points in an image and returns a pointer to the image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:
```
cip_buffer *cgr_line(image,x0,y0,x1,y1,gcw)
   cip_buffer *image;
   int x0, y0, x1, y1, gcw;
```

NOTES: *image:* points to the image in which the line is drawn.

x0, y0, x1, y1: the end points of the line and must lie within the image. If your image is *caq_image*, you can use the macros *CAQ_WIDTH* and *CAQ_HEIGHT* to compute an end point of a line, but you must use *(CAQ_WIDTH* - 1) and *(CAQ_HEIGHT* - 1) or the end point will be outside of the frame grabber image, since *CAQ_WIDTH* and *CAQ_HEIGHT* report the number of pixels across the frame grabber image and addressing starts with the location 0.

gcw: the graphics control word.

cgr_line() may be used to draw variations of shapes already covered by other Cognex graphics routines, such as a cross and an outline. For example, you can draw a cross of diagonal lines instead of horizontal and vertical lines; or you can draw an outline that is not rectangular, such as an arbitrary polygon.

cgr_line() also can be used on a frozen image. You can write routines to add dimension lines to processed images in the same way that a drafter adds lines to a technical drawing to indicate measurements. In this way an operator can identify important dimensions more quickly.

cgr_line() throws the following errors:

- *CIP_ERR_PELADDR* if either specified point lies outside the image or any of the specified coordinates is greater than 32,767
- *CIP_ERR_PELDEPTH* if the color value specified in the graphics control word cannot fit in the image's pixel size.

VARIABLE: **cgr_mod_dontcare**

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
int cgr_mod_dontcare
```

NOTES: *cgr_mod_dontcare* contains the value to use as the representation for a "don't care" pixel in the model. Values close to this value are not used to display ordinary model pixels. Suggested values for *cgr_mod_dontcare* are:

- 0 or 63 for grey models, so that "don't care" pixels show up as an extreme value while data pixels are substantially lighter or darker
- 32 for binary models, so that data pixels show up as black or white, while "don't care" pixels are an intermediate grey

cgr_mod_dontcare can be set at any time, but we recommend that you set it only at boot time since its value usually remains unchanged for a given environment. The default value of 0 is satisfactory for grey models.

FUNCTION: **cgr_outline()**

PURPOSE: Draws a border of specified value around the edges of an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_outline(image,gcw)
   cip_buffer *image;
   int gcw;
```

NOTES: **cgr_outline()** usually is used to indicate a region that is a window of a larger image. It is the suggested way to indicate a window of the frame grabber. It can also be used during model training to outline the area selected using the trackball interface. **cgr_outline()** returns a pointer to the image.

image: points to the image to be outlined.

gcw: the graphics control word.

Since an image is always rectangular, the outline drawn is always a rectangle equal in size to the image. It is drawn in the extreme edge rows and columns of the image. It does not affect anything outside of the image.

cgr_outline() throws the following errors:

- *CIP_ERR_PELADDR* if the image has less than two rows or less than one column
- *CIP_ERR_PELDEPTH* if the color value specified by the GCW is outside of *image*'s pixel depth

FUNCTION: **cgr_point()**

PURPOSE: Sets a pixel at the specified image coordinates to a new grey value and returns a pointer to the image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_point(image,x,y,z)
   cip_buffer *image;
   int x, y, z;
```

NOTES: *image:* points to the image in which the point is drawn.

x, y: the coordinates of the pixel.

z: the grey value to which to set the pixel.

cgr_point() cannot be drawn using the *CGR_PUSH* draw mode.

No errors are detected directly by **cgr_point()**. However, **cgr_point()** uses an image's **put()** and **get()** routines to access its pixels, and these routines can detect certain errors. For example, the error *CIP_ERR_PELADDR* is thrown if the specified pixel does not lie within the image. Also, the error *CIP_ERR_PELDEPTH* is thrown if the pixel value is unacceptable for the image type.

FUNCTION: **cgr_pointer()**

PURPOSE: Draws a pattern of a specified grey level in an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_pointer(image,x,y,gcw,style)
   cip_buffer *image;
   short x,y;
   int gcw;
   cgr_pointer_style style;
```

NOTES: There are five pointer icons: a large cross, a small cross, an "X", a circle, and a square. All icons are 5x5 pixels. **cgr_pointer()** draws extremely fast.

image: points to the image in which to draw the icon.

x, y: the coordinates of the center of the icon.

gcw: the graphics control word.

style: the type of icon to draw as specified by the following enumeration:

```
typedef enum{
   CGR_BIG_CROSS,
   CGR_SMALL_CROSS,
   CGR_BIG_X,
   CGR_BIG_SQUARE,
   CGR_BIG_CIRCLE
}  cgr_pointer_style;
```

cgr_pointer() throws the following errors:

- *CCHK_ERR_CHECK* if *style* is not one of the styles in the enumeration
- *CNOTIMP_ERR* if the image is not *FAST8*

ENUMERATION: **cgr_pointer_style**

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

DEFINITION:

```
typedef enum{
   CGR_BIG_CROSS,
   CGR_SMALL_CROSS,
   CGR_BIG_X,
   CGR_BIG_SQUARE,
   CGR_BIG_CIRCLE
}cgr_pointer_style;
```

NOTES: Specifies the style of the icon to create with **cgr_pointer().** All of the icons use the same number of pixels, but in the smaller version, some graphic pixels are drawn in the grey value of the image pixels being overwritten.

FUNCTION: **cgr_printf()**

PURPOSE: Draws characters for display on a monitor connected to the vision processor.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
int cgr_printf(sc,format,a,b,c,d,e,f,g,h,i,j)
   cgr_screen *sc;
   char *format;
   int a,b,c,d,e,f,g,h,i,j;
```

NOTES: Each character is a 5 x 9 dot matrix character with descending lower case. Each character is drawn in an 8 x 10 block of pixels. There is only one character drawing font built into the system (OCRA).

sc: points to the **cgr_screen** structure that defines the screen into which to draw the characters.

format: points to a format string, such as %d, indicating the format in which to print the characters.

a,b,c,d,e,f,g,h,i,j: optional arguments to the format string.

For more information on format strings and the optional arguments, refer to *C: A Reference Manual* by Harbison and Steele.

cgr_printf() should not be used with *CGR_PUSH* draw mode. Use **cgr_string()** to write characters when using *CGR_PUSH* draw mode; it is more efficient and can take advantage of the screen management features. If desired, you can use **sprintf()** to create a string and **cgr_string()** to draw the string.

Characters are laid out on a grid of *cellwd* by *cellht* blocks as specified by the **cgr_screen** data structure. Therefore, you cannot write characters at arbitrary pixel positions, as you can with the function **cgr_character()**. **cgr_printf()** maintains the current cursor position so that you need not calculate the position of each character in a string. **cgr_printf()** uses escape sequences to control the display. For example, you can position the cursor, make the cursor visible or invisible, and set the color of the screen background.

Cognex has defined a default **cgr_screen**, *cgr_fgscreen*, for writing characters to the frame grabber. The *cgr_fgscreen* default ink and background values, *scink* and *scbackgr*, are to be used with graphics overlaid on a real-time image; that is, you call **caq_graphics()** after the call to **cgr_printf()**. If you want to overlay graphics on a frozen image, you should change the values of *cgr_fgscreen->scink* and *cgr_fgscreen->scbackgr* as appropriate and call **caq_display()** after the call to **cgr_printf()**.

cgr_printf() uses the C run time library function, **sprintf()** to resolve a format string. The output of the resolved string is put into a buffer that is 512 characters long. The output contains not only the replacement of the format variables

by the corresponding values, but also the escape sequences; for example, the two characters "%d" are replaced with the corresponding decimal value. Be sure that you do not write past the end of the buffer. In general, a line of text about 80 characters long should not overflow the buffer. **cgr_printf()** returns an integer that is the number of characters sent to the output buffer by **sprintf()**.

Supported control characters are RETURN (\r), LINEFEED (\n), BACKSPACE (\b), ESCAPE (\033). All others are ignored.

Escape sequences, which are multiple character control functions, provide controls additional to the control characters in the ASCII 7-bit chart. These sequences are not displayed. They control character transmission, processing and display.

Escape sequences have the format:

ESC *<intermediate characters> <last character>*

Supported escape sequences are:

ESC [Px ; Py H	position cursor (\033[H homes cursor to upper left corner)
ESC [Px A	cursor up
ESC [Px B	cursor down
ESC [Px C	cursor left
ESC [Px D	cursor right
ESC [2 J	erase screen (\033[2J sets *scbackgr* to image)
ESC [Px m	set character attribute (0=normal 4=blink 7=inverse) (\033[0m sets *scink* to *ink*, and *scbackgr* to *backgr* \033[4m sets *scbackgr* to CAQ_BLINK \033[7m sets *scink* to *backgr*, and *scbackgr* to *ink)*
ESC [20 h	set newline mode (\n causes linefeed return)
ESC [20 l	clear newline mode (\n causes just linefeed)
ESC [? 5 h	set screen to white background, black foreground (sets *scink* and *ink* to CAQ_BLACK; sets *scbackgr* and *backgr* to CAQ_WHITE).
ESC [? 5 l	set screen to black background, white foreground (sets *scink* and *ink* to CAQ_WHITE; sets *scbackgr* and *backgr* to CAQ_BLACK).
ESC [? 25 h	set cursor visible

ESC [? 25 l set cursor invisible

Px and Py are optional numeric parameters, specifying number of character positions. These escape sequences are a subset of the escape sequences of a Digital Equipment Corporation VT100 terminal. However, the size of the vision processor screen (cgr_screen) differs from that of a VT100: the vision processor screen is 20 rows by 72 columns instead of 24 rows by 80 columns.

If you move the cursor past the right or left edge of the screen, it is placed at position one on the left edge of the screen. If you move the cursor past the bottom or top edge of the screen, it is placed at position one at the top of the screen.

Drawing and erasing the cursor is a very slow operation; you should make it visible only when prompting for input.

FUNCTION: **cgr_ruler_init()**

PURPOSE: Initializes a ruler figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

USE:
```
cgr_figure *cgr_ruler_init(fig)
   cgr_figure *fig;
```

NOTES: *fig*: points to a ruler figure. If *fig* is null, a ruler figure is allocated and a pointer to it is returned.

The following default general figure data is used when you initialize a ruler:

element	value
draw	cgr_ruler_draw
derive	cgr_ruler_derive
update	cgr_ruler_update
lbut	cif_std_lbut
mbut	cif_std_mbut
rbut	cgr_ruler_rbut
train	0
total_size	size_of(cgr_figure)
nm	2
z	CAQ_BLINK | CGR_PUSH
zp	CAQ_BLINK | CGR_PUSH
x0	caq_image->width/2
y0	caq_image->height/2
xres	0x100 (1)
yres	0x100 (1)

STRUCTURE: **cgr_screen**

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *screen.h*

DEFINITION:

```
typedef struct{
   cip_buffer **ibuff;
   cgr_vidfont *font;
   int cellwd, cellht,
      cursorx, cursory,
      flags;
   char
      ink, backgr, scink, scbackgr,
      cursorink,
      escargs, escarg[2];
} cgr_screen;
```

NOTES: **cgr_screen** defines the characteristics of the screen on which the graphics functions draw characters.

ibuff: points to a pointer to the **cip_buffer** containing the screen's grey values.

font: points to the **cgr_vidfont** structure containing the font of the characters to be written on the screen.

cellwd and *cellht:* the width and height, in screen pixels, of the cell or box in which each character is written. For example, by default **cgr_printf()** writes characters in cells that are 8 x 22 pixels. *cellwd* and *cellht* define a grid on which characters are written. Since characters in the default font are 8 x 20, there will be two blank rows of pixels between lines. By modifying *cellwd* and *cellht*, you can change the spacing between characters and the spacing between rows on the grid.

cursorx and *cursory:* the (x, y) coordinates (in screen coordinates from the upper left corner) of the next character to be drawn to the screen.

flags: controls the placement of characters on the screen. The flag bits are defined as follows:

cgr_sc_cursor: set if cursor visible

cgr_sc_newline: set if \n causes \n\r

cgr_sc_scroll: if one, scroll (not currently supported); if zero, go to top

cgr_sc_wrap: set to wrap when at right margin

cgr_sc_esc: currently receiving an escape sequence

cgr_sc_qerase: clear the screen to the current screen background color

ink and *backgr:* the *current* values for the characters' ink and background. The system defaults to *CAQ_BLACK* for *ink* and *CAQ_WHITE* for *backgr.* Available values are those values with up to 8 bits. You may obtain the default values using the escape sequences provided by **cgr_printf().** You must set other values directly.

If the image is a graphics mode image, *CAQ_BLACK, CAQ_WHITE, CAQ_PASS* and *CAQ_BLINK* are the only appropriate values.

scink and *scbackgr:* the default values for the characters' ink and background. The system defaults to *CAQ_BLACK* for *scink* and *CAQ_WHITE* for *scbackgr.* Available values are those values with up to 8 bits. For other values, you must set the value in the structure directly.

cursorink: the color of the cursor to be displayed on the screen. Values can be *CAQ_BLACK, CAQ_WHITE* and *CAQ_BLINK* or values having up to 8 bits. The default value is *CAQ_BLINK.*

escargs and *escarg[2]:* used in parsing the arguments to **cgr_printf()**.

escargs is either one or one (first or second of the arguments). *escargs[2]* is the array holding the two arguments. If you are modifying a **cgr_screen** data structure or creating a new one, all three elements should be set to zero. They are used only in parsing arguments after an escape.

For all practical purposes, the values of the following elements are fixed: *font=&std_font, cellwd=8, cellht=22.* The font is fixed because there are no tools for creating screen fonts. Changing *cellwd* and *cellht* results in increased spacing between characters, not in larger characters, when you use **cgr_printf()**.

The following elements are manipulated by **cgr_printf()** and you should not change them: *flags, escargs, escargs[0], escargs[1].*

You may change the values of the following elements directly: *cursorx, cursory, ink, backgr, scink, scbackgr, cursorink.*

The default ink and background values are for use with overlay graphics; i.e., a call to **caq_graphics()** after the call to **cgr_printf()**. To write on a frozen image, change the values of the ink and background to non-graphics values and follow the call to **cgr_printf()** with a call to **caq_display()**.

The variable *cgr_fgscreen* is the default **cgr_screen**. The values of the elements of *cgr_fgscreen* are as follows:

```
Element     value

**ibuff     &caq_image
*font       &std_font
cellwd      8
cellht      22
cursorx     0
cursory     0
flags       cgr_sc_newline|cgr_sc_wrap
ink         CAQ_BLACK
backgr      CAQ_WHITE
scink       CAQ_BLACK
scbackgr    CAQ_WHITE
cursorink   CAQ_BLINK
escargs     0
escarg[2]   0,0
```

VARIABLE: **cgr_stack**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

DEFINITION:
```
cgr_stack_header *cgr_stack;
```

NOTES: *cgr_stack* points to the current graphics stack.

cgr_stack_init() automatically sets *cgr_stack* to the new stack created.

FUNCTION: **cgr_stack_clear()**

PURPOSE: Re-initializes the current graphics stack.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
void cgr_stack_clear(stack)
   cgr_stack_header *stack
```

NOTES: A graphics stack must be cleared if you change the image in which you are drawing graphics. Note that no values are restored; the saved pixel values and frames are discarded. **cgr_stack_clear()** does not deallocate the graphics stack memory; the stack can be used again. **cgr_stack_clear()** does not erase graphics from the image.

stack: points to the graphics stack to clear.

FUNCTION: **cgr_stack_down()**

PURPOSE: Moves the stack top down one frame in the graphics stack.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
cgr_stack_down(stack,enable)
   cgr_stack_header *stack;
   int enable;
```

NOTES: When the stack top is moved down, the graphics object that is stored in the graphics frame that was previously at the top of the stack is erased from the image, but the frame information remains in stack memory.

stack: points to the graphics stack in which to move the stack top. The current graphics stack is represented by the global variable *cgr_stack*.

enable: indicates whether video synchronization (*CGR_VIDEO*) and graphics interrupts (*CGR_INTERRUPT*) should be enabled.

cgr_stack_down() throws the error *CGR_ERR_DOWN* if you are at the bottom of the stack.

STRUCTURE: **cgr_stack_frame**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

DEFINITION:

```
typedef struct {
   char
      color,
      state,
      group,
      id,
      *stack_p;
   int params[16];
   void
      (*draw) (),
      (*erase) (),
      (*change) ();
   struct cgr_stack_frame *down, *up;
}  cgr_stack_frame;
```

NOTES: **cgr_stack_frame** defines a graphics frame.

color: the grey value for the graphics frame item.

state: reserved for future use.

group: reserved for future use.

id: the ID for the graphics frame. This ID can be used to identify the graphics frame; it is ignored by Cognex software.

stack_p: points to the start of the save area associated with the graphics frame.

params: an integer array reserved for internal use.

draw: points to a routine for drawing the graphics associated with the graphics frame and saving the underlying image pixels to the save area.

erase: points to a routine for restoring the underlying image pixels, thereby erasing the graphics associated with the graphics frame.

change: points to a routine to redraw a graphic associated with the graphics frame without restoring or saving underlying pixel values. It is used to change the color of a primitive without having to pop and then push.

down: points to the graphics frame that was allocated before the current frame.

up: points to the graphics frame that was allocated after the current frame.

STRUCTURE: **cgr_stack_header**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

DEFINITION:

```
typedef struct {
   cgr_stack_frame
      *free,
      *top,
      *frames,
      bottom;
   char *save, *stack;
      int length;
}  cgr_stack_header;
```

NOTES: **cgr_stack_header** defines a graphics stack.

free: points to the free list for the stack.

top: points to the top of the stack.

frames: points to the start of the frames storage area in the graphics stack.

bottom: the bottom stack frame. The location of the bottom frame does not change.

save: points to the start of the stack area used for saving underlying image pixel values.

stack: points to the top of the stack area used for saving underlying image pixel values.

length: the number of graphics frames allocated for the stack.

FUNCTION: **cgr_stack_init()**

PURPOSE: Allocates memory for a graphics stack and creates the stack.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
cgr_stack_header *cgr_stack_init(frames,size);
   int frames, size;
```

NOTES: *frames*: the number of graphics frames to allocate for the graphics stack.

size: the memory size, in bytes, to allocate for the graphics stack.

The number of frames to allocate should be determined relative to the complexity and type of graphics you plan to display. All graphics primitives occupy a single graphics frame. Other graphics objects occupy multiple frames; the more complex the graphic, the more graphics frames used. A character overlay occupies a single graphics frame. Specify a graphics frame amount that is large enough to accommodate all the graphics reliably, but that is not excessive.

The size for the stack should be estimated according to the number of graphics to be displayed and the resolution of the image. The most efficient size may have to be derived empirically. If you specify a size and find that your graphics operations exceed stack limits, you can try increasingly larger allocations until you no longer run out of memory.

FUNCTION: **cgr_stack_pop()**

PURPOSE: Removes the graphics frame that is currently at the top of the graphics stack.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
void cgr_stack_pop(stack,enable)
   cgr_stack_header *stack;
   int enable;
```

NOTES: The graphics item stored in the popped frame is removed from the image and affected image pixels are restored to their original state.

stack: points to the graphics stack from which to remove the frame.

enable: indicates whether video synchronization (*CGR_VIDEO*) and graphics interrupts (*CGR_INTERRUPT*) should be enabled.

cgr_stack_pop() throws the error *CGR_ERR_POP* if a pop is attempted at the bottom of the stack.

FUNCTION: **cgr_stack_splice()**

PURPOSE: Moves a frame or group of frames to another location in the graphics stack.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
void cgr_stack_splice(stack, sourcetop, sourcebot, dest,
      enable)
   cgr_stack_header *stack;
   cgr_stack_frame *sourcetop, *sourcebot, *dest;
   int enable;
```

NOTES: *stack*: points to the graphics stack in which to move the frames.

sourcetop: points to the top frame of the group of frames you want to move.

sourcebot: points to the bottom frame of the group of frames you want to move.

dest: points to the frame above which you want to place the frames you are moving.

enable: indicates whether video synchronization (*CGR_VIDEO*) and graphics interrupts (*CGR_INTERRUPT*) should be enabled.

cgr_stack_splice() throws the error *CGR_ERR_SPLICE* if:

- *sourcetop*, *soucebot*, or *dest* are not frames in the stack specified by *stack.*
- The specified destination lies within the group of frames to be moved.
- The frame specified by *sourcetop* is not equal to or above the frame specified by *soucebot.*

FUNCTION: **cgr_stack_to_mark()**

PURPOSE: Moves the graphics stack top to the stack marker location by performing a specified stack function.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
int cgr_stack_to_mark(stack,function,mark,enable);
   cgr_stack_header *stack;
   void(*function)();
   cgr_stack_frame *mark;
   int enable;
```

NOTES: *stack*: points to the graphics stack (the global variable *cgr_stack* is ignored).

function: the stack function to perform in order to move the stack top. *function* can be *cgr_stack_pop*, *cgr_stack_down*, or *cgr_stack_up*.

mark: points to the stack marker.

enable: indicates whether video synchronization (*CGR_VIDEO*) and graphics interrupts (*CGR_INTERRUPT*) should be enabled.

cgr_stack_to_mark() performs the stack function specified by *function* repeatedly until the stack top reaches the stack marker location, or until it runs off an end of the stack. If the stack marker is found, 1 is returned; otherwise 0 is returned. Thus, if NULL is specified as the stack marker, you can move to the top or bottom of the stack without having to know the address of the frame.

FUNCTION: **cgr_stack_up()**

PURPOSE: Moves the stack top up one frame in the graphics stack.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

USE:

```
cgr_stack_up(stack,enable)
   cgr_stack_header *stack;
   int enable;
```

NOTES: When the stack top is moved up, the graphics object that is stored in the graphics frame that occupies the new stack top is redrawn in the image, and any underlying image pixel values are saved.

stack: points to the graphics stack in which to move the stack top.

enable: indicates whether video synchronization (*CGR_VIDEO*) and graphics interrupts (*CGR_INTERRUPT*) should be enabled.

cgr_stack_up() throws the error *CGR_ERR_UP* if there are no more frames above the top of the stack.

FUNCTION: **cgr_string()**

PURPOSE: Draws a character string in an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
void cgr_string(image,x,y,string,gcw,background,expand)
   cip_buffer *image;
   int x, y;
   char *string;
   int gcw, background, expand;
```

NOTES: Each character is a 5 x 9 dot matrix character with descending lower case. Each character is drawn in an 6 x 10 block of pixels. Optionally, the characters can be drawn in an 8 x 10 block of pixels, and can be doubled in x, y, or both.

image: points to the image in which to display the string. Typically, this is *caq_image*.

x, y: the image coordinates, in pixels, at which the upper left corner of the first character in the string is written. You can specify negative (x,y) coordinates; however, **cgr_string()** clips characters that are completely or partially outside the display area.

string: the null-terminated character string to display in the image. The maximum length of *string* is 1024 characters. String length may exceed the width of the display area; however, **cgr_string()** clips characters that are completely or partially outside the display area. **cgr_string()** does no formatting or parsing of characters in *string*.

gcw: the graphics control word.

background: specifies the background color for the string. The background for each character is any portion of the 8 x 10 block that is not occupied by the character. Specifying -1 leaves the background unchanged (transparent).

expand: indicates whether the height and/or width of the string characters should be altered from the standard eight pixels wide by 20 pixels high (doubled in y). *expand* can have one of the following values:

- *CGR_NO_EXP_CHAR_Y:* do not double the character height (the character will be drawn in a block that is 10 pixels high).
- *CGR_EXP_CHAR_X:* double both the character width and height. (12 pixels wide by 20 pixels high if *CGR_CELLWD_8* is clear, 16 pixels wide by 20 pixels high if *CGR_CELLWD_8* is set).
- *CGR_USE_Y_DOUBLING*: use the setting of the *CIP_FLAGS_NO_Y_DOUBLING* bit of the **cip_buffer** structure to determine expansion.

- *CGR_CELLWD_8*: draw the character in an 8 pixel wide block instead of the default 6 pixel wide block, or, if character doubling is used, draw the character in a 16 pixel wide block instead of the default 12 pixel wide block.

You can OR the *CGR_NO_EXP_CHAR_Y* and *CGR_EXP_CHAR_X* bits to double the width but not the height (12 pixels wide by 10 pixels high if *CGR_CELLWD_8* is clear, 16 pixels wide by 10 pixels high if *CGR_CELLWD_8* is set).

If an image is FAST8, **cgr_string()** only places characters at even addresses in the image. This is done for efficiency reasons. If an odd address is specified, **cgr_string()** subtracts one to create an even address. If this even forcing is undesirable for your application, you can disable the *CIP_FLAGS_FAST8* flag for the image so that the image is no longer FAST8. If an image is not FAST8, **cgr_string()** will place characters at even or odd addresses. However, character drawing is much slower if an image is not FAST8.

FUNCTION: **cgr_string_size()**

PURPOSE: Computes how large an area **cgr_string()** requires to draw the specified string.

INCLUDE FILE: *cgr.h*

USE:

```
void cgr_string_size (image, string, expand, w, h)
   cip_buffer *image;
   char *string;
   int expand;
   int *w, *h;
```

NOTES: **cgr_string_size()** interprets the *expand* argument of **cgr_string()** to compute how large an area is required to draw the string. This function is used for planning and formatting character strings in a display. For example, to draw a string centered on a point (x,y), you should use (x - *w*/2, y - *h*/2) as the starting point.

image: pointer to the destination image.

string: pointer to the null-terminated character string to display in the image.

expand: one of the constants which specify whether the height and/or width of the string characters should be altered from the standard eight pixels wide by 20 pixels high (doubled in y). See **cgr_string()** for a description of these values.

w and *h:* pointers to the width and height of the character string.

FUNCTION: **cgr_synthetic_closed_init()**

PURPOSE: Initializes a synthetic rectangle model figure that is to be trained as a model.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *ctr_syn.h*

USE:

```
cgr_figure *cgr_synthetic_closed_init(fig,p,h)
   cgr_closed *fig;
   cgr_synthetic_params *p;
   cgr_figure *h;
```

NOTES: *fig*: points to a synthetic rectangle model figure. If *fig* is null, a synthetic rectangle model figure is allocated and a pointer to it is returned.

p: points to the synthetic rectangle model figure parameters. If *p* is null, the figure parameters are set to the defaults as described for **cgr_closed_init()**.

h: points to the general figure data structure. If *h* is null, a general figure data structure is created with the default values specified for **cgr_closed_init()**, with the exception of the train structure parameter, which is set to the **ctr_synthetic_closed()** training function.

The default update speed settings are the same as used with **cgr_closed_init()**.

FUNCTION: **cgr_synthetic_corner_init()**

PURPOSE: Initializes a synthetic corner model figure that is to be trained.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *ctr_syn.h*

USE:

```
cgr_figure *cgr_synthetic_corner_init(fig,p,h)
   cgr_corner *fig;
   cgr_synthetic_params *p;
   cgr_figure *h;
```

NOTES: *fig*: points to a synthetic corner model figure. If *fig* is null, a synthetic corner model figure is allocated and a pointer to it is returned.

p: points to the synthetic corner model figure parameters. If *p* is null, the figure parameters are set to the defaults as described for **cgr_corner_init()**.

h: points to the general figure data structure. If *h* is null, a general figure data structure is created with the default values specified for **cgr_corner_init()**, with the exception of the *train* structure parameter, which is set to the **ctr_synthetic_corner()** training function.

The default update speed settings are the same as those used with **cgr_corner_init()**.

STRUCTURE: **cgr_synthetic_params**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *contour.h*

DEFINITION:

```
typedef struct {
   cia_point
      ap,
      mr,
      or,
      ir,
      mc,
      oc,
      ic;
   unsigned short b;
}  cgr_synthetic_params;
```

NOTES: **cgr_synthetic_params** defines various synthetic model parameters and is used by both synthetic corner model figures and synthetic rectangle model figures. These parameters must be supplied; they are not automatically derived.

ap: the anchor point of the figure, specified in physical units.

mr: the main contour radii of the figure, specified in physical units.

or: the outer contour radii of the figure, specified in physical units.

ir: the inner contour radii of the figure, specified in physical units.

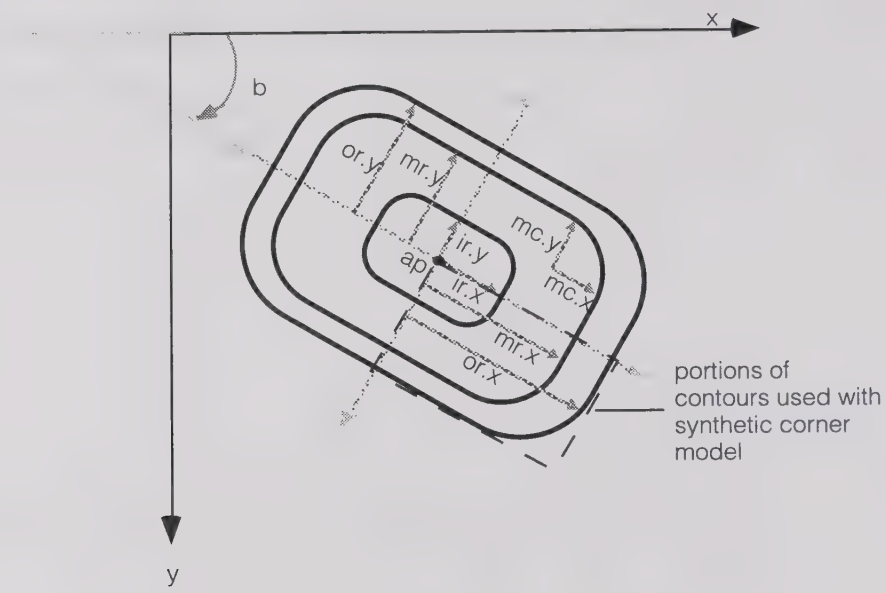
mc: the main corner radii of the figure, specified in physical units.

oc: the outer corner radii of the figure, specified in physical units.

ic: the inner corner radii of the figure, specified in physical units.

b: the angle of rotation of the figure, specified in binary. Rotation is clockwise.

The following diagram shows the relationship of **cgr_synthetic_params** elements to a synthetic rectangle model figure and a synthetic corner model figure.


x
b
or.y
mr.y
mc.y
ir.y
ap
ir.x
mc.x
mr.x
or.x
portions of
contours used with
synthetic corner
model
y

FUNCTION: **cgr_tone()**

PURPOSE: Writes alternating grey values to pixels in an image.

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *cgr.h*

USE:

```
cip_buffer *cgr_tone(image,z1,z2)
   cip_buffer *image;
   int z1,z2;
```

NOTES: Using **cgr_tone()**, you can produce a variety of effects, including a darkened or lightened image or a uniformly grey image. Each pixel in the image is classified as "even" or "odd." You specify two grey values to be written to the image: one to the "even" pixels and one to the "odd" pixels. The values you specify replace the current pixel values. The function returns a pointer to the modified image.

image: points to the image to which the new pixel values are written.

z1: the value written to "even" pixels in *image.*

z2: the value written to "odd" pixels in *image.*

If *z1* or *z2* is set to -1, the associated pixels are not written.

You can use **cgr_tone()** as an alternative to **cgr_outline()** to highlight a window of the frame grabber image. While **cgr_outline()** outlines the specified window, you can shade the entire window by calling **cgr_tone()** with *CAQ_PASS* and *CAQ_BLACK* or *CAQ_WHITE* as the alternating values. Alternating *CAQ_PASS* with *CAQ_BLACK* darkens the window, while alternating *CAQ_PASS* with *CAQ_WHITE* lightens the window.

You also can use **cgr_tone()** as an alternative to **cip_set()** to extend the effective choices of graphics colors for initializing an image background for display. By calling **cgr_tone()** with *CAQ_BLACK* and *CAQ_WHITE* as the alternating colors, the image is set to an intermediate grey.

This function is optimized for FAST8 images, although it may be used on images of any depth.

When used in graphics mode, **cgr_tone()** has the effect of "greying" the image.

In display mode, the function can be used to produce shading on top of an image: by alternating a grey level for *z1* or *z2* with -1 (i.e., by writing only to *z1* or *z2*), half of the original pixels will not be written to and therefore will be visible. **cgr_tone()** may also be used in display mode to alternate any two colors, although this is not a typical use.

In the case of images that have a RAT (the frame grabber image and images created by **cip_create()** or **cip_window()**), pixels are determined to be odd

or even in a way that is independent of windowing operations on the image. For an image with a RAT (e.g., *caq_image),* if a pixel is determined to be even and there is a call to **cgr_tone()**, that pixel will be set to *z1.* Then if **cgr_tone()** is called again on any window of that image that contains that pixel, the pixel will again be set to *z1.*

When **cgr_tone()** is called on a window of the image without a RAT (such as synthetic images where the pixels are provided by **get()** and **put()** routines), the pixels are not set to be the same (even or odd) as they are in the original image. In this case, an uneven pattern *may* result, depending on where the window falls on the original image.

"Odd" and "even" pixels are determined as follows:

- If pixel (*x,y*) is in image *ip* which has a RAT, then if the following expression evaluates to 0, pixel (*x,y*) is even. If the expression evaluates to one, pixel (*x,y*) is odd.

```
((((int) ip->rat) >>2 + ip->x_offset) +x + y) & 1
```

- If pixel (*x,y*) is in image *ip* which does not have a RAT, then if the expression evaluates to 0, pixel (*x,y*) is even. If the following expression evaluates to one, pixel (*x,y*) is odd.

```
(x + y) & 1
```

cgr_tone() throws the error *CGEN_ERR_BADARG:*

- if *image* is NULL
- if the width or height of *image* is less than zero
- if the depth of *image* is greater than *CIP_MAX_DEPTH*

VARIABLE: **cgr_video_start**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

DEFINITION:

```
int cgr_video_start;
```

NOTES: *cgr_video_start* indicates the first line in the range of video lines that are to be protected from graphics drawing. When video synchronization is enabled, no graphics drawing occurs between the time the line specified by *cgr_video_start* is displayed and the time the line specified by *cgr_video_stop* is displayed (unless a primitive takes longer to draw than the total time available in the unprotected zone, in which case the system will continue drawing during the protected period).

cgr_video_start is a number corresponding to a y coordinate value of the frame grabber in half resolution.

If *cgr_video_start* is grater than *cgr_video_stop*, no lines are protected. This may be useful in applications where the same code is used for both the 2000 and other Cognex products. In these cases, *CGR_VIDEO* can be set in the GCWs as appropriate for 2000 operation, and can be disabled on the 3000 and 4000 by setting *cgr_video_start* greater than *cgr_video_stop*.

VARIABLE: **cgr_video_stop**

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *gstack.h*

DEFINITION:

```
int cgr_video_stop;
```

NOTES: *cgr_video_stop* indicates the last line in the range of video lines that are to be protected from graphics drawing. When video synchronization is enabled, no graphics drawing occurs between the time the line specified by *cgr_video_start* is displayed and the time the line specified by *cgr_video_stop* is displayed (unless a primitive takes longer to draw than the total time available in the unprotected zone, in which case the system will continue drawing during the protected period).

cgr_video_stop is a number corresponding to a y coordinate value of the frame grabber in half resolution.

STRUCTURE: **cgr_vidfont**

REFERENCE: Image Processing: Graphics

INCLUDE FILE: *screen.h*

DEFINITION:

```
typedef struct{
   char *bits;
   int width, height;
   char lastchar;
} cgr_vidfont;
```

NOTES: **cgr_vidfont** defines a font for drawing characters on the screen.

bits: points to an array of type **char** having a size equal to:

height of font character * number of characters in font

width and *height:* describe the layout of the bits in *bits*; they specify the width and height of the characters in the font. For example, the characters drawn by the graphics functions use a font that has a width of 8 pixels and a height of 10 pixels.

Note: To avoid screen flicker, the vertical aspect of the characters is drawn twice; therefore the characters are drawn as 8 x 20 even though the value of *height* is 10.

lastchar: the ASCII code for the last character in the font. For example, \177 is the ASCII code for DELETE.

FUNCTION: **cgrz_circle()**

PURPOSE: Draws a circle.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_circle (tablet, x0, y0, r, z)
   cgrz_tablet *tablet;
   double
      x0, y0,
      r;
   int z;
```

NOTES: **cgrz_circle()** calls **cgr_ellipse()** to draw a line (it calls **cgr_ellipse()** instead of **cgr_circle()** so that it can use clipping).

The circle may not be visible at high zoom values because of the maximum boundary **cgr_ellipse()** places on the radius.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x0 and *y0:* the (x,y) coordinate of the center of the circle.

r: the radius of the circle.

z: the grey-scale value of the circle.

FUNCTION: **cgrz_coords()**

PURPOSE: Labels the top row and left column of full pixels with x and y source coordinates respectively.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_coords (tablet, zoom)
   cgrz_tablet *tablet;
   double zoom;
```

NOTES: **cgrz_coords()** labels the top row and left column of full pixels with x and y source coordinates respectively. The upper left pixel is not labeled. To maximize the contrast, the pixel value is drawn in *tablet->white* or *tablet->black*.

The labels are drawn only if *zoom* is no greater than the current zoom factor, and if the current zoom factor is larger than 40X, as expressed by the condition:

```
cgrz_get_zoom(tablet) >= zoom && cgrz_get_zoom(tablet)
>= 40
```

tablet: pointer to a previously initialized **cgrz_tablet** structure.

zoom: the zoom factor.

FUNCTION: **cgrz_copy()**

PURPOSE: Copies pixels from the source to the destination image.

INCLUDE FILE: *cgr_zoom.h*

USE:
```
void cgrz_copy (tablet)
   cgrz_tablet *tablet;
```

NOTES: Copies pixels from the source image to the destination image within the graphics tablet structure. It copies pixels (using **cip_copy()**) if the current zoom factor is equal to 1, expands pixels (using **cip_expand()**) if the current zoom factor is greater than 1, and samples pixels (using **cip_sample()**) if the current zoom factor is less than 1.

tablet: pointer to a previously initialized **cgrz_tablet** structure that contains pointers to the source and destination images *src* and *dst*, and the graphics source and destination overlay images *overlay_src* and *overlay_dst*.

If *overlay_src* is NULL, **cgrz_copy()** sets the *overlay_dst* image to *CAQ_PASS*, using **cip_set()**.

If *overlay_src* and *overlay_dst* are both non-NULL, **cgrz_copy()** uses **cgrz_scroll_zoom_copy()** to copy pixels from *overlay_src* to *overlay_dst* using the current zoom factor and scroll vector.

FUNCTION: **cgrz_ellipse()**

PURPOSE: Draws an ellipse or ellipse arc.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_ellipse (tablet, x0, y0, rx, ry, theta,
      phi1, phi2, z)
   cgrz_tablet *tablet;
   double
      x0, y0;
      rx, ry;
      theta;
      phi1, phi2;
   int z;
```

NOTES: **cgrz_ellipse()** calls **cgr_ellipse()** to draw an ellipse or ellipse arc. See **cgr_ellipse()** for an illustration which shows the relationship of the arguments to an ellipse arc.

The ellipse may not be visible at high zoom values because of the maximum bound **cgr_ellipse()** places on the radii.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x0 and *y0:* the (x,y) coordinate of the center of the ellipse.

rx and *ry:* the x and y radii of the ellipse. The maximum radius allowable is 2047.9375. The minimum radius allowable is 0.

theta: the angle of orientation of the ellipse (specified in degrees).

phi1 and *phi2:* the endpoints of the ellipse arc (specified in degrees). If *phi1* and *phi2* are equal, the ellipse arc is a closed ellipse.

z: the grey-scale value of the ellipse.

FUNCTION: **cgrz_fill()**

PURPOSE: Fills an area starting at the specified point.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_fill (tablet, x, y, z)
   cgrz_tablet *tablet,
   double x, y,
   int z;
```

NOTES: **cgrz_fill()** does a flood fill starting at the specified point. The fill is terminated by a boundary of pixels of the specified grey-scale value. The result may not be correct if the fill boundary is only partially visible in the destination image. **cgrz_fill()** calls **cgr_outline()** on the destination image to guarantee that a closed boundary exists.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x and *y:* the starting (x,y) point for the fill.

z: the grey-scale value of the boundary line.

FUNCTION: **cgrz_get_scroll()**

PURPOSE: Returns the current scroll vector.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_get_scroll (tablet, x0, y0)
   cgrz_tablet *tablet;
   double *x0, *y0;
```

NOTES: **cgrz_get_scroll()** returns the current scroll vector.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x0 and *y0:* the x and y coordinates of the source pixel being displayed at the center of the destination image.

FUNCTION: **cgrz_get_zoom()**

PURPOSE: Returns the current zoom factor.

INCLUDE FILE: *cgr_zoom.h*

USE:
```
double cgrz_get_zoom (tablet)
   cgrz_tablet *tablet;
```

NOTES: **cgrz_get_zoom()** returns the current zoom factor.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

FUNCTION: **cgrz_grid()**

PURPOSE: Draws a grid at pixel boundaries between the (magnified) displayed pixels.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_grid (tablet, zoom, z)
   cgrz_tablet *tablet;
   double zoom;
   int z;
```

NOTES: **cgrz_grid()** draws a grid at pixel boundaries between the (magnified) displayed pixels.

The grid is drawn only if *zoom* is no greater than the current zoom factor, and if the current zoom factor is larger than 2X, as expressed by the condition:

```
cgrz_get_zoom(tablet) >= zoom && cgrz_get_zoom(tablet) > 2
```

tablet: pointer to a previously initialized **cgrz_tablet** structure.

zoom: the zoom factor.

z: the grey-scale value of the grid lines.

FUNCTION: **cgrz_line()**

PURPOSE: Draws a line.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_line (tablet, x0, y0, x1, y1, z)
   cgrz_tablet *tablet;
   double
      x0, y0,
      x1, y1;
   int z;
```

NOTES: **cgrz_line()** calls **cgr_iline()** to draw a line.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x0 and *y0:* the starting (x,y) coordinate of the line.

x1 and *y1:* the ending (x,y) coordinate of the line.

z: the grey-scale value of the line.

VARIABLE: **cgrz_overrides**

INCLUDE FILE: *cgr_zoom.h*

DEFINITION:

```
cgrz_overrides_struct cgrz_overrides;
```

NOTES: *cgrz_overrides* is a global variable that facilitates external control of the optional scroll/zoom loop graphics. *cgrz_overrides* is of type **cgrz_overrides_struct**:

```
typedef struct {
   cgrz_overview_mode overview_mode;
   char
      do_pelvalues;
      do_grid;
      do_coords;
} cgrz_overrides_struct;
```

Setting any of the following components to a non-default value forces the corresponding graphics to be drawn in all **cgrz_scroll_zoom()** loops.

overview_mode: the method in which the overview is to be drawn. It can take one of the values in the **cgrz_overview_mode** enumeration. The default value is *cgrz_overview_none*.

do_pelvalues: a flag that specifies whether to display the numerical grey-scale values in each (magnified) pixel. A non-zero value specifies enable, and 0 specifies disable. The default value is 0.

do_grid: a flag that specifies whether to display a grid at pixel boundaries between the (magnified) pixels. A non-zero value specifies enable, and 0 specifies disable. The default value is 0.

do_coords: a flag that specifies whether to label the top row and left column of (magnified) pixels with x and y source coordinates respectively. A non-zero value specifies enable, and 0 specifies disable. The default value is 0.

FUNCTION: **cgrz_overview()**

PURPOSE: Displays a reduced copy of the source image with a graphic box representing the current view in the destination image.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_overview (tablet, overview_mode)
   cgrz_tablet *tablet;
   cgrz_overview_mode overview_mode;
```

NOTES: **cgrz_overview()** draws an overview, which is a reduced source image with a graphic box representing the current view into the destination image. The overview's aspect ratio is the same as that of the source image; it is never larger than 1/4 the size of the destination image in either dimension.

The overview is drawn only if the current zoom factor is larger than the minimum, as expressed by the condition:

```
cgrz_get_zoom(&tablet) > tablet.min_s
```

tablet: pointer to a previously initialized **cgrz_tablet** structure containing pointers to the source and destination images.

overview_mode: the method in which the overview is to be drawn. It can take one of the values in the **cgrz_overview_mode** enumeration.

cgrz_overview() is normally called automatically by **cgrz_scroll_zoom()**.

ENUMERATION: **cgrz_overview_mode**

INCLUDE FILE: *cgr_zoom.h*

DEFINITION:

```
typedef enum {
   cgrz_overview_none,
   cgrz_overview_float,
   cgrz_overview_ul,
   cgrz_overview_ur,
   cgrz_overview_bl,
   cgrz_overview_br
} cgrz_overview_mode;
```

NOTES: **cgrz_overview_mode** specifies the method in which the overview is to be drawn.

cgrz_overview_none specifies that the overview not be drawn.

cgrz_overview_float specifies that the overview location float, i.e., avoid the active trackball area.

cgrz_overview_ul specifies that the overview be drawn in the upper left corner of the destination image.

cgrz_overview_ur specifies that the overview be drawn in the upper right corner of the destination image.

cgrz_overview_bl specifies that the overview be drawn in the bottom left corner of the destination image.

cgrz_overview_br specifies that the overview be drawn in the bottom right corner of the destination image.

FUNCTION: **cgrz_pelvalues()**

PURPOSE: Draws the pixel values in each (magnified) displayed pixel.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_pelvalues (tablet, zoom)
   cgrz_tablet *tablet;
   double zoom;
```

NOTES: **cgrz_pelvalues()** displays the numerical grey-scale values in each (magnified) pixel. To maximize the contrast, the grey-scale pixel value is drawn in *tablet->white* or *tablet->black*.

The pixel values are drawn only if the value specified for *zoom* is no greater than the current zoom factor, and if the current zoom factor is larger than 20X, as expressed by the condition:

```
cgrz_get_zoom(tablet) >= zoom && cgrz_get_zoom(tablet)
>= 20
```

tablet: pointer to a previously initialized **cgrz_tablet** structure.

FUNCTION: **cgrz_point()**

PURPOSE: Draws a point.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_point (tablet, x0, y0, z)
   cgrz_tablet *tablet;
   double x0, y0;
   int z;
```

NOTES: **cgrz_point()** calls **cgr_iline()** to draw a line (it calls **cgr_iline()** instead of **cgr_point()** so that it can use *CGR_PUSH*).

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x0 and *y0:* the (x,y) coordinate of the point.

z: the grey-scale value of the point.

FUNCTION: **cgrz_pointer()**

PURPOSE: Draws a pointer.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_pointer (tablet, x, y, z, style)
   cgrz_tablet *tablet;
   double x, y;
   char z;
   short style;
```

NOTES: **cgrz_pointer()** calls **cgr_pointer()** to draw a small pointer or marker centered on the specified point.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x and *y:* the (x,y) coordinate of the pointer.

z: the grey-scale value of the pointer.

style: the type of icon to use as the pointer, as specified by the **cgr_pointer_style** enumeration:

```
typedef enum {
   CGR_BIG_CROSS,
   CGR_SMALL_CROSS,
   CGR_BIG_X,
   CGR_BIG_SQUARE,
   CGR_BIG_CIRCLE
} cgr_pointer_style;
```

FUNCTION: **cgrz_scroll_relative()**

PURPOSE: Attempts to change the scroll vector from its current position by the specified number of pixels.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_scroll_relative (tablet, dx, dy)
   cgrz_tablet *tablet;
   double dx, dy;
```

NOTES: **cgrz_scroll_relative()** attempts to change the scroll vector from (*x0,y0*) to (*x0+dx,y0+dy*). It is constrained in same way as **cgrz_set_scroll()**.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

dx and *dy:* the x and y change in the scroll vector, in pixels.

FUNCTION: **cgrz_scroll_zoom()**

PURPOSE: Scrolls and zooms based on incoming trackball data.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
int cgrz_scroll_zoom (tablet)
   cgrz_tablet *tablet;
```

NOTES: **cgrz_scroll_zoom()** reads the trackball and calls **cgrz_scroll_relative()**, **cgrz_zoom_in()**, **cgrz_zoom_out()**, **cgrz_copy()**, and **cgrz_overview()** as appropriate.

cgrz_scroll_zoom() returns 0 when the left trackball button is pressed; it returns 1 when no trackball input is available and the graphics need to be updated. The middle button calls **cgrz_zoom_out()**. The right button calls **cgrz_zoom_in()**. Trackball x-coordinate data *dx* is negated (reversed) when passed to **cgrz_scroll_relative()**.

Note that you must first initialize *cif_tb_file* with a call to **cif_open_tb()**.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

FUNCTION: **cgrz_scroll_zoom_copy()**

PURPOSE: Initializes a graphics tablet structure and copies pixels from the source image to the destination image.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_scroll_zoom_copy (src, dst, x0, y0, zoom)
   cip_buffer *src, *dst;
   double
      x0, y0,
      zoom;
```

NOTES: **cgrz_scroll_zoom_copy()** initializes a temporary graphics tablet structure, then copies pixels from the source image to the destination image within the graphics tablet structure. It copies pixels (using **cip_copy()**) if the current zoom factor is equal to 1, expands pixels (using **cip_expand()**) if the current zoom factor is greater than 1, and samples pixels (using **cip_sample()**) if the current zoom factor is less than 1.

Calling **cgrz_scroll_zoom_copy()** is equivalent to calling **cgrz_setup()** and **cgrz_copy()**.

cgrz_scroll_zoom_copy() requires that *src* and *dst* be non-NULL and not overlap.

src: pointer to the source image.

dst: pointer to the destination image.

x0 and *y0:* the x and y coordinates of the source pixel to be displayed at the center of the destination image. For example, *x0* = 5 and *y0* = 6 specifies that the source pixel (5,6) be displayed at the center of the destination image. *x0* and *y0* are currently truncated to the nearest integer.

zoom: the zoom factor.

cgrz_scroll_zoom_copy() throws *CGEN_ERR_BADARG* if:

- *src* or *dst* is NULL
- *dst* is smaller than 2 in either dimension

FUNCTION: **cgrz_set_scroll()**

PURPOSE: Sets the scroll vector.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_set_scroll (cgrz_tablet, x0, y0)
   cgrz_tablet *tablet;
   double x0, y0;
```

NOTES: **cgrz_set_scroll()** sets the scroll vector.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x0 and *y0:* the x and y coordinates of the source pixel to be displayed at the center of the destination image.

x0 and *y0* are clipped to the range bounded by (0,0) and (*src->width*, *src->height*).

FUNCTION: **cgrz_set_zoom()**

PURPOSE: Sets the zoom factor.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_set_zoom (tablet, zoom)
   cgrz_tablet *tablet;
   double zoom;
```

NOTES: **cgrz_set_zoom()** sets the zoom factor.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

zoom: the new zoom factor. It is rounded to the nearest integral power of two, and is constrained to the interval specified by the *min_s* and *max_s* components of the *cgrz_tablet* structure.

min_s is the largest integral power of two that permits the entire source image to be viewed in the destination image, or 1, whichever is smaller. Note that you can specify a fractional zoom factor such as 1/4 or 1/16.

max_s is the largest integral power of two that permits two full source image pixels to be drawn in each dimension of the destination image.

FUNCTION: **cgrz_setup()**

PURPOSE: Initializes a graphics tablet structure.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_setup (tablet, src, dst, x0, y0, zoom)
   cgrz_tablet *tablet;
   cip_buffer *src;
   cip_buffer *dst;
   double x0, y0;
   double zoom;
```

NOTES: **cgrz_setup()** initializes the specified graphics tablet structure. It sets the following components of the **cgrz_tablet** structure to the indicated default values:

- *white* = 255
- *black* = 0
- *overview_mode* = *cgrz_overview_br*
- *overlay_src* = NULL
- *overlay_dst* = NULL

In addition, the following read-only components are set to the indicated values:

- *min_s* is set to the largest power of 2 which is less than or equal to 1 and which allows the entire source image to be seen in the destination.
- *max_s* is set to the largest power of 2 which is greater than or equal to 1 and which allows four full source image pixels to be drawn in the destination.

cgrz_setup() requires that *src* and *dst* be non-NULL and not overlap. *dst* should be a display image or a window into one.

tablet: pointer to the **cgrz_tablet** structure to initialize.

src: pointer to the source image.

dst: pointer to the destination image.

x0 and *y0:* the x and y coordinates of the source pixel to be displayed at the center of the destination image. For example, *x0* = 5 and *y0* = 6 specifies that the source pixel (5,6) be displayed at the center of the destination image. *x0* and *y0* are currently truncated to the nearest integer.

zoom: the zoom factor.

cgrz_setup() throws *CGEN_ERR_BADARG* if:

- *src* or *dst* is NULL
- *dst* is smaller than 2 in either dimension
- *tablet* is NULL

FUNCTION: **cgrz_string()**

PURPOSE: Displays the specified text string.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_string (tablet, x, y, string, z, background,
      expand)
   cgrz_tablet *tablet;
   double x, y;
   char *string;
   int z, background, expand;
```

NOTES: **cgrz_string()** calls **cgr_string()** to draw a text string in the destination image. See the description of **cgr_string()** for more details.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

x and *y:* the (x,y) coordinates at which the upper left corner of the first character in the string is written. You can specify negative (x,y) coordinates; however, **cgr_string()** clips characters that are completely or partially outside the display area.

string: pointer to the null-terminated character string to display in the image. The maximum length of *string* is 1024 characters. String length may exceed the width of the display area; however, **cgr_string()** clips characters that are completely or partially outside the display area.

z: the grey-scale value of the text string.

background: the background color for the string. The background for each character is any portion of the 8 x 10 block that is not occupied by the character. Specifying -1 leaves the background unchanged (transparent).

expand: one or more of the ORable constants listed below, which specify whether the height and/or width of the string characters should be altered from the standard eight pixels wide by 20 pixels high (doubled in y). See the description of **cgr_string()** for more details on these values.

CGR_NO_EXP_CHAR_Y: Do not double character height.

CGR_EXP_CHAR_X: Double both the character width and height.

CGR_USE_Y_DOUBLING: Use the setting of the **cip_buffer** bit *CIP_FLAGS_NO_Y_DOUBLING* to determine expansion.

CGR_CELLWD_8: Draw the character in an 8 pixel wide block, or in a 16 pixel wide block if doubling is used.

STRUCTURE: **cgrz_tablet**

INCLUDE FILE: *cgr_zoom.h*

DEFINITION:

```
typedef struct cgrz_tablet {
   int white;
   int black;
   cgrz_overview_mode overview_mode;
   cip_buffer
      *overlay_src,
      *overlay_dst,
      *src,
      *dst;
   double
      min_s, max_s;
} cgrz_tablet;
```

NOTES: The **cgrz_tablet** structure contains the data that defines the graphics tablet. Its components are user-settable unless otherwise noted.

white: the pixel value that draws white.

black: the pixel value that draws black.

overview_mode: the method in which the overview is to be drawn. It can take one of the values in the **cgrz_overview_mode** enumeration.

overlay_src: pointer to the graphics overlay source image, i.e., an image from which graphics are to be copied to *overlay_dst*. Pixels are resized (expanded or reduced) according to the current zoom factor.

In the default case, *overlay_src* is NULL, and is ignored. *overlay_src* should be non-NULL only if you have a valid *overlay_dst*; if you have not specified *overlay_dst*, no graphics are drawn.

overlay_src is useful for region graphics (e.g. graphics that highlight a region of defect pixels by setting them to *CAQ_BLINK*), because the graphics are resized with the region. *overlay_src* is not recommended for geometric graphics, such as lines, circles, boxes, etc., for which resizing is undesirable.

The image specified for *overlay_src* must have the same dimensions and alignment as that of *src*.

overlay_dst: pointer to the graphics overlay destination image, i.e., an image to which graphics are to be drawn.

In the default case, *overlay_dst* is NULL, and graphics are drawn on the destination image.

If *overlay_dst* is non-NULL, graphics may be drawn either on *overlay_dst* or *dst*. A scroll zoom graphics function draws on *overlay_dst* if the con-

stant *CGR_OVERLAY* is ORed into the function's grey-scale value argument. If not, the function draws graphics on *dst*.

The image specified for *overlay_dst* must have the same dimensions and alignment as that of *dst*.

src: pointer to the source image. This component is read only; you must specify the source image using **cgrz_setup()**.

dst: pointer to the destination image. This component is read only; you must specify the destination image using **cgrz_setup()**.

min_s and *max_s:* indicate the minimum and maximum scale (zoom factor) permitted. These components are read only; you cannot set their values.

The scroll zoom graphics functions refer to the graphics tablet for information on how to perform their respective graphics operations. The functions adhere to the following conventions:

- Spatial points and dimensions are specified in source image pixel coordinates. Source image coordinates are transformed to destination image coordinates according to the current scroll and zoom setting of the graphics tablet.

  Note: Although the scroll and zoom settings are part of the graphics tablet, they are not user-accessible. You cannot directly set these components; rather, you must use the following functions as appropriate to set or change their values:

  cgrz_set_scroll()

  cgrz_set_zoom()

  cgrz_scroll_relative()

  cgrz_zoom_in()

  cgrz_zoom_out()

  cgrz_scroll_zoom()

- Angles are specified in degrees.

- The scroll zoom graphics functions draw graphics using the grey-scale value specified by the *z* argument. If *overlay_dst* is NULL, and *CGR_OVERLAY* is ORed into the *z* argument, the grey-scale value (*z* && 0xFF) is drawn to *dst* and is mapped as follows:

  CAQ_WHITE corresponds to the *white* component of the graphics tablet.

  CAQ_BLACK corresponds to the *black* component of the graphics tablet.

  CAQ_BLINK corresponds to the *white* component of the graphics tablet.

CAQ_PASS corresponds to the *black* component of the graphics tablet.

Any other value causes the function to throw *CGEN_ERR_BADARG*.

FUNCTION: **cgrz_zoom_in()**

PURPOSE: Attempts to zoom in by a factor of two.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_zoom_in (tablet)
   cgrz_tablet *tablet;
```

NOTES: **cgrz_zoom_in()** attempts to zoom in by a factor of two. It is constrained in the same way as **cgrz_set_zoom()**.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

FUNCTION: **cgrz_zoom_out()**

PURPOSE: Attempts to zoom out by a factor of two.

INCLUDE FILE: *cgr_zoom.h*

USE:

```
void cgrz_zoom_out (tablet)
   cgrz_tablet *tablet;
```

NOTES: **cgrz_zoom_out()** attempts to zoom out by a factor of two. It is constrained in the same way as **cgrz_set_zoom()**.

tablet: pointer to a previously initialized **cgrz_tablet** structure.

STRUCTURE: **cgtc_acq**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   short gain, offset;
} cgtc_acq;
```

NOTES: **cgtc_acq** is reserved for your use. You can store two acquisition parameters, camera gain and offset (described in the *Acquisition and Display* chapter of the *Image Processing* manual) that can be used by your application code. Both parameters default to -1, which indicates they are not used. (3000/4000/5000 only.)

gain: the camera gain.

offset: the camera offset.

FUNCTION: **cgtc_blob()**

PURPOSE: Computes the geometric properties of blobs in an image (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cblob_results *cgtc_blob(dm,bimg,rp,td,bp,results,rip_pp)
   cgtc_module *dm;
   cip_buffer *bimg;
   cgtc_run_params *rp;
   cgtc_tmpl_data *td;
   cblob_params *bp;
   cblob_results *results;
   crip_buffer **rip_pp;
```

NOTES: GTC uses **cgtc_blob()** to compute the geometric properties of blobs in an image. Blob segmentation results are computed from the binary filtered error image. The grey value blob information is either computed from the difference directly or from the grey-scale, non-thresholded, error image. The latter method involves the additional step of computing the error grey-scale image.

The GTC software performs blob analysis on defect pixels using the function **cgtc_blob()** which calls **cblob_zero()**, **cblob_one()**, or **cblob_two()**. Blob analysis is controlled by the components *max_area* and *blob_flags* of the structure **cgtc_run_params**. See **cgtc_run_params** for a full description of these parameters.

dm: pointer to the defect module to process.

bimg: pointer to the binary image to use for blob analysis.

rp: pointer to the run-time parameters; the *blob_flags* component specifies how blob statistics are computed.

td: pointer to the template data structure.

bp: pointer to the blob parameters.

The pixel weighting function *bp.wf* is set to *cmap_absolute.*

The absolute linear mapping parameters *bp.mp.ab.zero* and *bp.mp.ab.one* indicate break points that are used in computing the internal arbitrary mapping used in **cgtc_blob()**. If either of the two components is set to -1, an internally computed value is used.

results: pointer to the blob results structure. If *results* is NULL, **cgtc_blob()** allocates the blob results structure, and you are responsible for freeing it using **cgtc_delete_results()**. If you allocate the blob results structure using **cu_alloc()**, the structure must be zeroed out using **cu_clear** before the first time it is called.

rip_pp: an array of pointers to the blob segmentation results reported in the form of a **crip_buffer** structure if *rip_pp* is non-null. If *rip_pp* is NULL, the segmentation results are discarded.

cgtc_blob() returns a pointer to the blob results structure.

FUNCTION: **cgtc_copy_module()**

PURPOSE: Creates a windowed copy of a specified defect module (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cgtc_module *cgtc_copy_module (dm, wdm, xorig, yorig,
      wd, ht)
   cgtc_module
      *dm, *wdm;
   int
      xorig, yorig,
      wd, ht;
```

NOTES: **cgtc_copy_module()** creates a windowed copy of a specified defect module called the *parent module.* All the images in the windowed copy are windowed copies of corresponding images or global images in the parent module, and the window statistics are recomputed. The search model in the windowed copy is a copy of the search model in the parent window.

cgtc_copy_module() works the same way as **cgtc_window_module()** except that it creates copies of images instead of windows of images. That is, the images in the windowed copy are snapshots of images in the parent defect module, and are not affected if the parent defect module is deleted.

dm: pointer to the parent defect module.

wdm: pointer to the windowed copy of the defect module. If *wdm* is NULL, **cgtc_window_module()** creates a new window defect module.

xorig and *yorig:* the upper left corner of the windows relative to the images in the parent module.

wd and *ht:* the width and height of the window. If *wd* or *ht* is greater than the available width or height, the window is truncated to the maximum size possible.

cgtc_copy_module() throws the following errors:

- *CGTC_ERR_PARAMS* if *xorig* or *yorig* or both exceed the bounds of the image in the parent module.

FUNCTION: **cgtc_delete_global()**

PURPOSE: Deletes the global buffers (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_delete_global()
```

FUNCTION: **cgtc_delete_module()**

PURPOSE: Deallocates (deletes) a defect module (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:
```
void cgtc_delete_module (dm)
   cgtc_module *dm;
```

NOTES: *dm:* pointer to the defect module to be deleted.

FUNCTION: **cgtc_delete_mresults()**

PURPOSE: Deletes a multiple inspection result structure.

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

USE:
```
void *cgtc_delete_mresults (mres)
   cgtc_mresults *mres;
```

NOTES: **cgtc_delete_mresults()** deletes a multiple inspection result structure.

You should use **cgtc_delete_mresults()** to delete **cgtc_mresults** structures which have been created by **cgtc_meta_run()**. You can also use **cgtc_delete_mresults()** to delete structures you have allocated off the stack. If you allocate a structure off the stack, you must call **cu_clear()** on the structure before using it.

mres: pointer to the multiple inspection result structure to be deleted.

FUNCTION: **cgtc_delete_results()**

PURPOSE: Deletes a GTC results structure (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:
```
void cgtc_delete_results(res)
   cgtc_results *res;
```

NOTES: *res:* pointer to the GTC results structure to delete.

VARIABLE: **cgtc_dflt_map**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
extern unsigned char cgtc_dflt_map[256];
```

NOTES: *cgtc_dflt_map:* the current default normalization map. It is implemented as an array of 256 elements. **cgtc_run()** derives a normalization mapping based on a comparison of the statistics of the test image and the template image, and stores this map in *cgtc_dflt_map*. (3000/4000/5000 only.)

VARIABLE: **cgtc_dimg**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:
```
extern cip_buffer *cgtc_dimg;
```

NOTES: *cgtc_dimg:* pointer to the difference image (an 8-bit, global **cip_buffer** the size of the frame grabber). (3000/4000/5000 only.)

VARIABLE: **cgtc_eimg**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:
```
extern cip_buffer *cgtc_eimg;
```

NOTES: *cgtc_eimg:* pointer to the error image (a global **cip_buffer** the size of the frame grabber; 1 bit deep in binary morphology, or 8 bits deep in grey-scale morphology). (3000/4000/5000 only.)

VARIABLE: **cgtc_fimg**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:
```
extern cip_buffer *cgtc_fimg;
```

NOTES: *cgtc_fimg:* pointer to the error filtered image (a global **cip_buffer** the size of the frame grabber, 1 bit deep in binary morphology, or 8 bits deep in grey-scale morphology). (3000/4000/5000 only.)

STRUCTURE: **cgtc_geom**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   cip_buffer
      *image,
      *insp_wind,
      *reg_wind;
   short
      mpx, mpy;
} cgtc_geom;
```

NOTES: **cgtc_geom** contains all the window information for the inspection and search model windows. It is created before the defect module, and is used in the defect module for the alignment of images during training. (3000/4000/5000 only.)

The V code creates and initializes the inspection and registration windows with default sizes and positions, then passes these window figures to **cif_track_figure()**. **cif_track_figure()** allows you to modify the size and position of the inspection and registration windows using a Cognex trackball device. See the V code for an example of how this is done.

image: pointer to the acquired (frame grabber) image.

insp_wind: pointer to the inspection window.

reg_wind: pointer to the search model window.

mpx, mpy: the x and y coordinates, respectively, of the manual alignment point.

FUNCTION: **cgtc_init_global()**

PURPOSE: Allocates the global buffers **cgtc_dimg**, **cgtc_eimg**, and **cgtc_fimg** (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_init_global (image, type)
   cip_buffer *image;
   cgtc_mod_type type;
```

NOTES: *image:* pointer to a base image used in the current application. The global buffers are the same size as *image*.

type: the type of buffer that is created.

CGTC_TYPE_BIN: creates a 1-bit deep error and filtered error image.

CGTC_TYPE_GREY: creates an 8-bit deep grey-scale error and filtered error image.

FUNCTION: **cgtc_load_vb_action()**

PURPOSE: Loads the graphics action routine.

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_load_vb_action()
```

NOTES: *Note:* **cgtc_load_vb_action()** has been superseded by **caq_load_vb_action()**. Cognex will continue to support **cgtc_load_vb_action()** to maintain upward compatibility. You should use **caq_load_vb_action()** in any new code you write.

cgtc_load_vb_action() loads the graphics action routine, which executes at vertical blanking (3000/4000/5000 only). The action routine allows full graphics capability on frozen images without using the graphics plane *caq_gplane*.

FUNCTION: **cgtc_make_gwindow()**

PURPOSE: Reconstructs the global windows *dimg*, *eimg*, and *fimg* of the specified defect module (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_make_gwindow (dm)
   cgtc_module *dm;
```

NOTES: **cgtc_make_gwindow()** reconstructs the global windows *dimg*, *eimg*, and *fimg* of the specified defect module. *dimg*, *eimg*, and *fimg* are windows of the global images *cgtc_dimg*, *cgtc_eimg*, and *cgtc_fimg*.

You can delete the global images *cgtc_dimg*, *cgtc_eimg*, and *cgtc_fimg* using **cgtc_delete_global()** and can re-create them using **cgtc_init_global()**. However, if you create a module while the global images are NULL, the global windows *dimg*, *eimg*, and *fimg* will be NULL. **cgtc_make_window()** recomputes and sets the global windows.

dm: the defect module containing the global buffer windows *dimg*, *eimg*, and *fimg*.

FUNCTION: **cgtc_make_timg()**

PURPOSE: Computes, allocates, and returns an array of threshold images for multiple GTC inspection.

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

USE:

```
cip_buffer **cgtc_make_timg (dm, thr_cst, thr_slp, index,
      mask)
   cgtc_module *dm;
   int thr_cst;
   float thr_slp;
   int index;
   cip_buffer *mask;
```

NOTES: **cgtc_make_timg()** computes and allocates an array of threshold images from the standard deviation images contained in the **cgtc_module** structure and from user-defined parameters.

To reduce the overall time required for inspection, Meta GTC can optionally compute a set of threshold images before run time.

Note: You are responsible for freeing the threshold images array correctly: use **cip_delete()** to delete the **cip_buffer** threshold images, then **free()** to free the array.

dm: pointer to the defect module.

thr_cst: the threshold constant.

thr_slp: the threshold slope.

index: the index of the subpixel bin to use. To specify all bins, set *index* to -1, in which case **cgtc_make_timg()** creates all the relevant subpixel bin threshold images.

mask: pointer to an optional mask image that will be included in the threshold image. Refer to the *mask* component of the **cgtc_module** structure for a detailed description.

cgtc_make_timg() returns an array of pointers to precalculated threshold images.

FUNCTION: **cgtc_map_sobel()**

PURPOSE: Maps the Sobel image to the standard deviation image. (3000/4000/5000 only.)

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_map_sobel (dm, sobel, cst, slope, index)
   cgtc_module *dm;
   cip_buffer *sobel;
   double cst, slope;
   int index;
```

NOTES: **cgtc_map_sobel()** maps the Sobel image to the standard deviation image, according to the equation *sdimg*[i] = *slope* * *Sobel*[i] + *cst*.

dm: pointer to the defect module.

sobel: pointer to the Sobel image.

cst, *slope:* the constant and slope of the linear equation.

index: an integer specifying which subpixel bin of *sdimg* to map. -1 specifies all bins.

cgtc_map_sobel() throws the following errors:

- *CGTC_ERR_NOSOBIMG* if *sobel* is not a valid *cip_buffer*
- *CGEN_ERR_BADARG* if *dm* or *dm->spa* is NULL

FUNCTION: **cgtc_mask_xfer()**

PURPOSE: Directly transfers the mask image to the threshold image (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_mask_xfer (mask, dst, maskval)
   cip_buffer *mask, *dst;
   int maskval;
```

NOTES: **cgtc_mask_xfer()** directly transfers the mask image to the threshold image. If a mask pixel is set to 1, the corresponding pixel in the destination image is unchanged; if a mask pixel is set to 0, the corresponding pixel in the destination image is set to the value of *maskval*.

mask: pointer to a 1-bit mask image.

dst: pointer to an 8-bit image to which the mask is transferred.

maskval: the value to which to set masked pixels (i.e. destination image pixels that correspond to don't care pixels in the mask image.

FUNCTION: **cgtc_meta_run()**

PURPOSE: Performs a multiple GTC inspection.

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

USE:

```
cgtc_mresults *cgtc_meta_run (img, dm, mrp, sr, mres,
      opnum)
   cip_buffer *img;
   cgtc_module *dm;
   cgtc_mrun_params *mrp;
   cse_results *sr;
   cgtc_mresults *mres;
   int *opnum;
```

NOTES: **cgtc_meta_run()** performs a multiple inspection using an array of inspection parameter sets.

cgtc_meta_run() reorders the inspection computations to reduce the number of intermediate results calculations. The internal reordering minimizes the number of times that the threshold image must be recomputed from the standard deviation image and that the mask must be transferred onto the threshold image. The reordering also groups together all morphological filters associated with a given set of threshold parameters.

cgtc_meta_run() calculates and returns a set of numerical results for each set of inspection parameters provided, along with the statistics and resultant images from the last inspection. The individual results are returned as a linked list of result structures whose order does not necessarily correspond to that of the input array. The individual inspection results are identified by the thresholding parameters *thr_cst* and *thr_slp*, and by the morphological function *morph_func* in the **cgtc_res_lst** structure.

img: pointer to the whole-pixel aligned test image to inspect.

dm: pointer to the defect module.

mrp: pointer to the multiple run-time inspection parameters.

sr: pointer to the search results, which are used to perform the alignment of *img*.

mres: pointer to the multiple inspection result structure to be returned.

If *mres* is NULL, **cgtc_meta_run()** allocates the multiple results structure, and you are responsible for freeing it using **cgtc_delete_mresults()**.

opnum: pointer to an integer that records the number of operations performed by the multiple inspection software. In the integer *opnum*, the 10000's digit indicates the number of threshold computations, the 1000's digit indi-

cates the number of inspections, and the remainder indicates the number of morphological filtering operations.

For example, an *opnum* value of 13002 signifies 1 threshold computation, 3 inspections, and 2 morphological filtering operations.

STRUCTURE: **cgtc_mod_params**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   cgtc_trn_params tp;
   void
      (*dflt_morph)(),
      (*dflt_gmorph)();
} cgtc_mod_params;
```

NOTES: **cgtc_mod_params** contains parameters that are needed to create the defect module. (3000/4000/5000 only.)

tp: structure containing the training parameters.

dflt_morph: pointer to the default morphological function. If you specify a NULL pointer for *dflt_morph*, GTC uses **cgtc_erode_3x3()**.

dflt_gmorph: pointer to the default grey-scale morphological function. If you specify a NULL pointer for *dflt_gmorph*, GTC uses **cgtc_grey_erode_c3x3()**.

STRUCTURE: **cgtc_module**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   int id;
   int nv_flags;
   cgtc_trn_params tp;
   short
      apx, apy,
      mpx, mpy,
      ix0, iy0,
      iwd, iht,
      maxn;
   int *sums_tab;
   unsigned char *map;
   cip_buffer
      *dimg,
      *eimg,
      *fimg,
      *mask,
      *fmask;
   void (*morph_func)();
   void (*gmorph_func)();
   cgtc_tmpl_data *spa;
   cse_model **model;
   cip_buffer *sobel;
   cip_lreg_results *linreg;
   cgtc_acq *acq_params;
   int user_params[2];
   int reserved;
} cgtc_module;
```

NOTES: **cgtc_module** holds the information that makes up the defect module. It is created and initialized using a call to the function **cgtc_new_module()**, and is updated by successive training runs, which are executed using the function **cgtc_train()**. (3000/4000/5000 only.)

When a defect module is created, you can specify that subpixel alignment be used (see the components *spx* and *spy* in the structure **cgtc_trn_params**).

The defect module contains separate template, standard deviation, and threshold images for each subpixel phase of the template. The subpixel bins are implemented as an array of subpixel templates of type *cgtc_tmpl_data*; this array is pointed to by the component *spa*, described below.

id: a 32-bit integer ID. The lower 16 bits are an ID assigned to the defect module when it is created. The upper 16 bits contain the training state of the defect module. If the defect module has been trained with a VC-2 copro-

cessor, the upper 16 bits are set to *CGTC_ID_TRN_VC2*; otherwise the upper 16 bits are set to 0.

nv_flags: flag that indicates which of the global pointers or buffers are not valid. If **receive_module()** cannot reconstruct a pointer to one or more of the global pointers or buffers, it sets the value of *nv_flags* to one or more of the following:

CGTC_NV_MAP: global mapping is not valid

CGTC_NV_DIMG: global difference image window is not valid

CGTC_NV_EIMG: global error image window is not valid

CGTC_NV_FIMG: global filtered error image window is not valid

tp: structure containing a copy of the training parameters **cgtc_trn_params**.

apx, *apy:* the frame grabber x and y coordinates of the upper-left-most pixel of the search model window.

mpx, *mpy:* the frame grabber x and y coordinates of the manual alignment point.

ix0, *iy0:* the frame grabber x and y coordinates of the image origin (upper left corner) of the inspection window.

iwd, *iht:* the width and height of the inspection window.

maxn: the maximum number of samples per subpixel bin. The maximum number of samples depends on the number of bits used in the training images, as set by the *bits* component of the **cgtc_trn_params** structure.

sums_tab: pointer to a summation table used by the training step to add samples quickly to the accumulator image.

map: pointer to the computed pixel map, which is used for image normalization.

dimg: pointer to a window of the global buffer **cgtc_dimg** (the difference image). The window has the size of the inspection window, and the global buffer has the size of the frame grabber.

Pixels in the difference image that represent a negative difference are mapped to the value *CAQ_WHITE*, and pixels that represent a positive difference are mapped to the value *CAQ_BLACK*.

eimg: pointer to a window of the global buffer **cgtc_eimg**, which is the size of the frame grabber, 1 bit deep in binary morphology, or 8 bits deep in grey-scale morphology.The window has the size of the inspection window, and the global buffer has the size of the frame grabber.

fimg: pointer to a window of the global buffer **cgtc_fimg**, which is the size of the frame grabber, 1 bit deep in binary morphology, or 8 bits deep in

grey-scale morphology. The window has the size of the inspection window, and the global buffer has the size of the frame grabber.

mask: pointer to a 1-bit mask image the same size as the difference image. Each pixel of the mask image corresponds to a pixel in the difference image. In the mask image, a value of 0 specifies don't care (that the pixel never be a defect pixel) and a value of 1 specifies care (that the pixel be processed normally).

The mask image is automatically transferred into the threshold image when you create a new module using **cgtc_new_module()** or change the threshold parameters using **cgtc_set_ti_thresh()**. You may also transfer the mask image directly using **cgtc_mask_xfer()**.

If you do not want to use masking, specify a mask image of all ones.

fmask: pointer to the 1-bit filtered mask image. Because you perform morphological filtering on the error image, the mask image must be filtered in the same way in order to match the features of the filtered error image. You obtain the filtered mask image from the mask image by filtering the mask image with the *same* morphological structuring element that you use to structure defects.

Note: You must specify *fmask* by calling **cgtc_set_morph()**, which performs the necessary procedure to obtain a filtered mask image.

morph_func: pointer to the morphological function to be used in the morphology step. Note that the only legal way to set *morph_func* is by calling **cgtc_set_morph()**. If you specify a NULL pointer for *morph_func*, GTC uses the morphological function specified in the *dflt_morph* component of the **cgtc_mod_params** structure. The binary GTC morphological functions are listed in the *GTC* chapter of the *Vision Tools* manual.

gmorph_func: pointer to the grey-scale GTC morphological function to be used in the morphology step. It is used only if you have specified the grey-scale morphological inspection method. Unlike *morph_func*, you can set *gmorph_func* directly. If you specify a NULL pointer for *gmorph_func*, GTC uses the grey-scale GTC morphological function specified in the *dflt_gmorph* component of the **cgtc_mod_params** structure. The grey-scale GTC morphological functions are listed in the *GTC* chapter of the *Vision Tools* manual.

spa: pointer to an array of subpixel templates of type *cgtc_tmpl_data*. The array is of size *spx* by *spy*; see the structure **cgtc_trn_params**.

model: pointer to an array of pointers to search models (**cse_model**) that are used to align all images during training and inspection. The array of search models is of size **num_cse_mod**, and is defined when the defect module is created.

sobel: pointer to the Sobel magnitude image.

linreg: pointer to a structure of type **cip_lreg_results**, which contains the results of the linear regression between the Sobel image and the standard deviation image. **cip_lreg_results** is described in the release notes for this document. **cgtc_train()** provides values only for the regression coefficients *a* and *b* (slope and constant), as these are the only values necessary for the linear regression calculation. The remaining parameters are set to NULL. *linreg* is used in one-sample training.

acq_params: pointer to the acquisition parameters contained in the structure **cgtc_acq**.

user_params: two integers reserved for your use.

reserved: reserved for future use.

FUNCTION: **cgtc_morph()**

PURPOSE: Performs the binary morphological filtering of the source image (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cip_buffer *cgtc_morph(src,dst,mask,function,limit,count)
   register cip_buffer
      *src, *dst,
      *mask;
   void (*function)();
   int limit, *count;
```

NOTES: **cgtc_morph()** first performs morphological filtering, masks the result (if you have specified a mask image), then counts the remaining defect pixels.

src: pointer to the binary source image.

dst: pointer to the destination image. If *dst* is NULL, then **cgtc_morph()** allocates a destination image *dst*, and you are responsible for deallocating it.

mask: pointer to an optional 1-bit mask image (aligned binary image) in which a value of 0 specifies don't care (that the pixel never be a defect pixel) and a value of 1 specifies care (that the pixel be processed normally).

function: pointer to the morphological function to apply to the source image. The morphological functions are listed with their corresponding invalid data regions in the *GTC* chapter of the *Vision Tools* manual. You are responsible for windowing and coordinate translation in dealing with the invalid data regions.

If you do not want to perform any morphology operation, use the identity function **cgtc_morph_ident()**:

```
extern void cgtc_morph_ident()
```

For dilation and erosion functions whose structuring element has an even size, there is a reverse function (a mirror image of the structuring element, symmetric along the x and y axes) used to compensate for the off-center origin. The reverse functions have the same name as the direct functions except they have an "r" appended to the name. For example, the reverse function for **cgtc_erode_2x3()** is **cgtc_erode_2x3r()**. The invalid data regions for the reverse structuring elements are also listed with their corresponding invalid data regions in the *GTC* chapter of the *Vision Tools* manual. Some structuring elements are symmetric; in the case of a symmetric function, the reverse function is redundant and is not provided.

limit: counting limit for defect pixels. A value of 0 specifies no counting limit.

count: pointer to the count of defect pixels after morphological filtering. If *count* is non-NULL, **cgtc_morph()** returns the number of ones in the result image, up to the value of *limit*. *Note*: Because the result image is binary, ones represent defect pixels.

STRUCTURE: **cgtc_mpar**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

DEFINITION:

```
typedef struct {
   short thr_cst;
   float thr_slp;
   cip_buffer **timg;
   short
      limit,
      nblob,
      min_area,
      max_area;
   void (*morph_func)();
} cgtc_mpar;
```

NOTES: **cgtc_mpar** specifies an individual set of inspection parameters.

If you specify a value of -1 for any of the numerical parameters (*thr_cst*, *thr_slp*, *limit*, *nblob*, *min_area* or *max_area*), **cgtc_meta_run()** uses the value of the corresponding component in the global **cgtc_mrun_params** structure:

thr_cst, *thr_slp*, *limit* and *max_area* default to the corresponding value in the *rp* component of the **cgtc_mrun_params** structure.

nblob and *min_area* default to the values specified in the *blobp* component of the **cgtc_mrun_params** structure.

thr_cst: the constant component of the threshold image. The constant component is *b* in the linear equation $y = ax + b$, where x is the standard deviation image and y is the resulting threshold image.

thr_slp: the sigma (slope) component of the threshold image. The slope is *a* in the linear equation $y = ax + b$, where x is the standard deviation image and y is the resulting threshold image.

timg: pointer to an array of threshold images. The number of elements in the array corresponds to the number of subpixel bins. To speed up the calculation of threshold images, you can construct an array of precomputed threshold images using **cgtc_make_timg()** and specify the returned array in *timg*. Each **cgtc_mpar** inspection parameter set can have a different precomputed threshold image. If *timg* is NULL, the inspection software computes a threshold image for each inspection that specifies a change in the threshold parameters.

limit: the limit at which the blob analysis is to stop counting defect pixels.

nblob: the number of blob results records returned by blob analysis (for more details, see the *results* component of the **cblob_params** structure).

min_area: the minimum total area (pixel weight of defect pixels) that is to be considered a blob in blob analysis (see the *min_area* component of the **cblob_params** structure).

max_area: the maximum total area (pixel weight of defect pixels) to be considered a blob in blob analysis. If the biggest blob has an area greater than *max_area*, the blob computation is aborted and no blob results are reported.

morph_func: pointer to the morphological function to use in the morphology step. The morphological functions are listed in the *GTC* chapter of the *Vision Tools* manual. If you specify a NULL pointer for *morph_func*, Meta GTC uses the morphological function currently specified in the defect module.

Meta GTC supports binary GTC morphological operations only.

The mask and filtered mask images are derived from the defect module (*mask* and *fmask* components of the **cgtc_module** structure); however, you can provide your own mask images using **cgtc_make_timg()**.

Note: In Meta GTC applications, you should not use **cgtc_set_morph()** to specify the morphological function. Rather, you should directly specify the *morph_func* component of the **cgtc_mpar** structure.

STRUCTURE: **cgtc_mresults**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

DEFINITION:

```
typedef struct {
   int id;
   short spx, spy;
   cgtc_run_meth meth;
   char *norm_func;
   cse_results *sr;
   cgtc_tmpl_data *td;
   cip_buffer
      *dimg,
      *eimg,
      *fimg;
   int time[10];
   cgtc_res_lst *res;
} cgtc_mresults;
```

NOTES: **cgtc_mresults**contains the global and individual results of the multiple inspection. The *res_form* component of the **cgtc_run_params** structure indicates which optional results appear in the global results structure **cgtc_mresults**. Note that *dimg*, *eimg*, and *fimg* point to images left in the state of the last inspection.

The individual results for each inspection are reported in the linked list *res*.

Note: You should delete the **cgtc_mresults** structure by using **cgtc_delete_mresults()**.

id: the integer ID assigned to the defect module when it is created.

spx and *spy:* the number of subpixel bins in x and y.

meth: the run-time method for which the defect module was trained.

norm_func: pointer to the normalization function used. *norm_func* is used if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_NFUN;* if not, *norm_func* is set to NULL.

sr: pointer to the search results (defined if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_SR;* if not, *sr* is set to NULL).

td: pointer to template data of the appropriate subpixel resolution. (It is used if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_TD;* if not, *td* is set to NULL.)

dimg: pointer to the difference image from the last inspection performed. (It is used if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_GBUF;* if not, *dimg* is set to NULL.)

eimg: pointer to the error image from the last inspection performed. (It is used if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_GBUF;* if not, *eimg* is set to NULL.)

fimg: pointer to the filtered error image from the last inspection performed. (It is used if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_GBUF;* if not, *fimg* is set to NULL.)

time: an array of execution times, expressed in number of ticks. A tick is a unit of time equal to the line frequency (63.5 µs in RS-170). To convert from ticks to ms, multiply the number of ticks by the line frequency (i.e., 0.0635 ms for RS-170). See the *time* component of the **cgtc_results** structure for a detailed description of execution times.

res: pointer to a linked list of individual inspection result structures.

STRUCTURE: **cgtc_mrun_params**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

DEFINITION:

```
typedef struct {
   cgtc_run_params rp;
   cblob_params blobp;
   int mpar_num;
   cgtc_mpar *mpar;
} cgtc_mrun_params;
```

NOTES: **cgtc_mrun_params** contains all the run-time parameters for multiple inspection.

The inspection parameters are specified in an array of **cgtc_mpar** structures and in a set of global inspection parameters contained in a **cgtc_run_params** structure. Each **cgtc_mpar** structure specifies the parameters that define a single inspection. Changing any of the threshold parameters or the morphological filter defines a new inspection. The **cgtc_run_params** structure specifies all other inspection parameters. If any parameter in the **cgtc_mpar** structure is not specified, it defaults to the value of the corresponding component in the **cgtc_run_params** structure or the **cblob_params** structure.

In addition, the **cgtc_mrun_params** structure contains a set of blob inspection parameters. The **cblob_params** structure provides parameters for the overall blob analysis, and also provides default values for some of the parameters in the **cgtc_mpar** structure.

rp: the global run-time parameters. See the **cgtc_run_params** structure for a full description of all its components.

blobp: the blob analysis parameters.

mpar_num: the number of inspection parameter sets (i.e., the number of elements in the array pointed to by *mpar*). The maximum number is limited by memory.

mpar: pointer to an array of inspection parameter sets. Any parameter not specified in *mpar* defaults to the value of the corresponding component in *rp*.

FUNCTION: **cgtc_new_module()**

PURPOSE: Creates a defect module (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cgtc_module *cgtc_new_module (image, ddm, modp, geom,
      model, type)
   cip_buffer *image;
   cgtc_module *ddm;
   cgtc_mod_params *modp;
   cgtc_geom *geom;
   cse_model *model;
   cgtc_mod_type type;
```

NOTES: **cgtc_new_module()** initializes the defect module according to how you set the parameter *type*, which has three options: binary full GTC module, binary constant intensity GTC module, and grey full GTC module.

image: pointer to the base image (in frame buffer or RAM).

ddm: pointer to a valid defect module from which the sample and search model windows, and the search model are copied. If *ddm* is NULL, training data is obtained from the window structure pointed to by *geom* and the search model pointed to by *model*. If *ddm* is non-NULL, and you have not specified a geometry *geom*, you must first call **cgtc_init_global()** at system start-up time. Once you call **cgtc_init_global()**, you do not need to call it again every time you call **cgtc_new_module()**.

modp: pointer to parameters needed to create the defect module.

geom: pointer to window and alignment information.

model: pointer to a search model (**cse_model**) that is used to align all images during training and inspection if *ddm* has been set to NULL.

type: the module type, it is defined according to the following enumeration.

```
typedef enum{
   CGTC_TYPE_BIN,
   CGTC_TYPE_GREY,
   CGTC_TYPE_CST
} cgtc_mod_type;
```

CGTC_TYPE_BIN: Create a binary defect module for full GTC.

CGTC_TYPE_GREY: Create a grey defect module for full GTC.

CGTC_TYPE_CST: Create a binary defect module for constant intensity GTC.

FUNCTION: **cgtc_receive_module()**

PURPOSE: Retrieves a defect module from a designated storage location (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtcxfer.h*

USE:

```
cgtc_module
*cgtc_receive_module(name,input_fn,input_params)
   char *name;
   c_function input_fn;
   char *input_params;
```

NOTES: **cgtc_receive_module()** retrieves a defect module from a storage location such as a host computer or nonvolatile memory. It returns a pointer to the defect module.

Before retrieving a defect module with **cgtc_receive_module()**, you must first call **cgtc_init_global()** at system start-up time. Once you call **cgtc_init_global()**, you do not need to call it again every time you call **cgtc_receive_module()**.

name: a filename passed to the I/O function during initialization. This is usually used as a filename but may be NULL. For a complete description of how *name* is processed see the *I/O* chapter of the *System Software* manual.

input_fn: pointer to a function to receive the defect module from hardware. You can write your own function or use one of the Cognex input functions. Refer to the *I/O* chapter in the manual, *System Software*, for more information about the Cognex input functions.

If you are using your own input function, you can pass data to this function with the *input_params* pointer.

input_params: pointer to parameters to be passed to the input function. If you are using a Cognex input function, then the input parameters are specified by a parameters structure specific to the type of transfer taking place. Refer to the *I/O* chapter of the *System Software* manual for detailed information on I/O parameters structures.

cgtc_receive_module() throws the following errors:

- *CGHT_ERR_BADARG* if any argument is not valid
- *CGHT_ERR_PARAMS* if a field of a parameters structure is not valid

STRUCTURE: **cgtc_res_lst**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc_met.h*

DEFINITION:

```
typedef struct {
   short thr_cst;
   float thr_slp;
   char *morph_func;
   int raw_cnt;
   int mor_cnt;
   int num_blob;
   cblob_results *bres;
   struct cgtc_res_lst *next;
} cgtc_res_lst;
```

NOTES: **cgtc_res_lst** contains a linked list of individual result structures for each inspection, as well as the most prominent inspection parameters.

thr_cst: the constant component of the threshold image. The constant component is *b* in the linear equation $y = ax + b$, where x is the standard deviation image and y is the resulting threshold image.

thr_slp: the sigma (slope) component of the threshold image. The slope is *a* in the linear equation $y = ax + b$, where x is the standard deviation image and y is the resulting threshold image.

morph_func: pointer to the name of the morphology function (used if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_MFUN;* if not, *morph_func* is set to NULL).

raw_cnt: the total count of defect pixels before the morphology step.

mor_cnt: is the total count of defect pixels after the morphology step.

num_blob: is the number of blobs identified.

bres: pointer to the results of the blob analysis.

next: pointer to the next element in the linked list.

FUNCTION: **cgtc_reset_vb_action()**

PURPOSE: Disables the graphics action routine (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:
```
void cgtc_reset_vb_action()
```

NOTES: *Note:* **cgtc_reset_vb_action()** has been superseded by **caq_reset_vb_action()**. Cognex will continue to support **cgtc_reset_vb_action()** to maintain upward compatibility. You should use **caq_reset_vb_action()** in any new code you write.

cgtc_reset_vb_action() disables the graphics action routine previously enabled using the function **cgtc_load_vb_action()**.

STRUCTURE: **cgtc_results**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   int
      id,
      raw_cnt, mor_cnt;
   cgtc_run_meth meth;
   char *morph_func, *norm_func;
   short thr_cst;
   float thr_slp;
   short spx, spy;
   int time[10];
   int num_blob;
   cblob_results *bres;
   cse_results *sr;
   cgtc_tmpl_data *td;
   cip_buffer *dimg, *eimg, *fimg;
}  cgtc_results;
```

NOTES: **cgtc_results** contains the inspection results. It contains information that may be useful to the interpretation or decision-making process; however, note that some of the components of **cgtc_results** are pointers to intermediate calculations, and thus may not be useful. (3000/4000/5000 only.)

raw_cnt: total count of defect (error) pixels before morphology step

mor_cnt: total count of defect (error) pixels after morphology step

num_blob: number of discrete blobs identified

bres: pointer to the results of the blob analysis. See the *Inspection* chapter of the *Vision Tools* manual for more information on blob results.

id: the ID of the corresponding GTC module.

raw_cnt: the defective pixel count, before morphology.

mor_cnt: the defective pixel count, after morphology.

meth: the run-time method for which the defect module was trained.

morph_func: pointer to the morphology function (used if *cgtc_run_params.res_form = CGTC_RES_MFUN;* if not, *morph_func* is set to NULL).

norm_func: pointer to the normalization function (used if *cgtc_run_params.res_form = CGTC_RES_NFUN;* if not, *norm_func* is set to NULL).

thr_cst: the threshold constant.

thr_slp: the threshold slope.

spx, spy: the number of subpixel bins in x and y.

time: an array of execution times (in number of ticks). A tick is a unit of time equal to the line frequency (63.5 µs in RS-170). To convert from ticks to ms, multiply the number of ticks by the line frequency (i.e., 0.0635 ms for RS-170).

time[0]: total time

time[1]: registration

time[2]: grey-scale normalization curve computation

time[3] depends on the inspection method:
CGTC_INSP_ST: normalization, subtraction, abs. value, thresholding
CGTC_INSP_TI: normalization
CGTC_INSP_CST: mean and standard deviation computation
CGTC_INSP_GREY: normalization

time[4] depends on the inspection method:
CGTC_INSP_TI: absolute value, comparison, thresholding
CGTC_INSP_CST: constant GTC
CGTC_INSP_GREY: grey-scale threshold subtraction

time[5] depends on the inspection method:
CGTC_INSP_ST: filter, mask, count
CGTC_INSP_TI: filter, mask, count
CGTC_INSP_CST: filter, mask, count
CGTC_INSP_GREY: grey-scale filter

time[6]: blob statistics computation

time[7-9]: reserved

num_blob: the number of computed blobs.

bres: pointer to the results of the blob analysis.

sr: pointer to the search results (defined if *cgtc_run_params.res_form* = *CGTC_RES_SR;* if not, *sr* is set to NULL).

td: pointer to template data of the appropriate subpixel resolution (used if *cgtc_run_params.res_form* = *CGTC_RES_TD;* if not, *td* is set to NULL).

dimg: pointer to the difference image (used if *cgtc_run_params.res_form* = *CGTC_RES_GBUF;* if not, *dimg* is set to NULL).

eimg: pointer to the error image (used if *cgtc_run_params.res_form* = *CGTC_RES_GBUF;* if not, *eimg* is set to NULL).

fimg: pointer to the filtered error image (used if *cgtc_run_params.res_form = CGTC_RES_GBUF;* if not, *fimg* is set to NULL).

STRUCTURE: **cgtc_rt_trn_params**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   cgtc_trn_opts trn_meth;
   cip_edge_params *edge_p;
}  cgtc_rt_trn_params;
```

NOTES: **cgtc_rt_trn_params** contains parameters used in the training of sample images. (3000/4000/5000 only.)

trn_meth: enumeration that controls which mode of training **cgtc_train()** performs:

```
typedef enum {
   CGTC_TRN_SAMP,
   CGTC_TRN_COPY,
   CGTC_TRN_STAT,
   CGTC_TRN_COMP,
   CGTC_TRN_RUN,
   CGTC_TRN_SOB,
   CGTC_TRN_MPSOB,
   CGTC_TRN_LREG,
   CGTC_TRN_CST
} cgtc_trn_opts;
```

CGTC_TRN_SAMP: Acquire and add one sample image to the accumulator image.

CGTC_TRN_COPY: Copy the sample image directly into the template image.

CGTC_TRN_STAT: Compute only the statistics necessary to normalize subsequent input training samples (histogram, tails, standard deviation).

CGTC_TRN_COMP: Compute all the statistics needed for run-time inspection, based on the current training database, but retain the accumulator image so further training is possible.

CGTC_TRN_RUN: Compute all the statistics needed for run-time inspection but flush the accumulator image so no further training is possible.

CGTC_TRN_SOB: Compute the Sobel magnitude of source image.

CGTC_TRN_MPSOB: Map the Sobel magnitude to standard deviation.

CGTC_TRN_LREG: Compute the linear regression between the standard deviation image and the Sobel magnitude image.

CGTC_TRN_CST: Train for constant intensity GTC. Compute the mean and standard deviation of the source image and store these values in the *mean* and *sd* components of the *spa* component of the defect module.

edge_p: pointer to the Sobel edge detection parameters. For a description of the **cip_edge_params** structure, see the *Edge Detection Tool* chapter in the *Image Processing* manual.

FUNCTION: **cgtc_run()**

PURPOSE: Implements the run-time part of the GTC algorithm (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cgtc_results *cgtc_run (img, dm, rp, sr, res, gtc_blob)
   cip_buffer *img;
   cgtc_module *dm;
   cgtc_run_params *rp;
   cse_results *sr;
   cgtc_results *res;
   cblob_params *gtc_blob;
```

NOTES: When you call **cgtc_run()**, the system internally checks whether the defect module was trained on a different platform: with or without a VC-2 coprocessor. (The *id* component of the **cgtc_module** structure contains this information.) If so, GTC remaps the threshold image and *id* is reset accordingly. The remapping only occurs the first time you call **cgtc_run()**.

img: pointer to the whole-pixel aligned test image to process.

dm: pointer to the defect module.

rp: pointer to the run-time inspection parameters.

sr: pointer to the search results, which are used to perform the subpixel alignment of *img*.

res: pointer to the GTC **cgtc_results** structure to be returned.

If *res* is NULL, **cgtc_run()** allocates the GTC results structure, and you are responsible for freeing it using **cgtc_delete_results()**.

gtc_blob: pointer to the blob parameters structure used in blob analysis. For a description of **cblob_params**, see the *Inspection* chapter of the *Vision Tools* manual.

cgtc_run() throws the following errors:

- *CGTC_ERR_NOTMPL* if the inspection method requested is single threshold (*CGTC_INSP_ST*) or threshold image (*CGTC_INSP_TI*), and the defect module *dm* contains template image that is NULL or number of samples equals 0
- *CGTC_ERR_NOTIMG* if the inspection method requested is threshold image (*CGTC_INSP_TI*), and the threshold image *timg* is NULL
- *CGTC_ERR_INVMETH* if the inspection method requested by *rp.insp_meth* is not one of the valid methods defined in the enumeration *cgtc_insp_opts* (see the structure **cgtc_run_params**)

- *CGTC_ERR_NOGWIND* if one of the global windows (*dimg*, *eimg*, or *fimg*) of the defect module is NULL. Use **gtc_init_global()** to set the global windows.

FUNCTION: **cgtc_run_cst()**

PURPOSE: Performs constant intensity GTC (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cgtc_results *cgtc_run_cst (img, dm, rp, res, gtc_blob,
      msigma)
   cip_buffer *img;
   cgtc_module *dm;
   cgtc_run_params *rp;
   cgtc_results *res;
   cblob_params *gtc_blob;
   int msigma;
```

NOTES: There are two methods of performing constant intensity GTC: one computes a threshold based on a single trained image, and the other computes a threshold based only on the run-time image.

If the *trn_meth* component of the **cgtc_rt_trn_params** structure is set to *CGTC_TRN_CST*, and an image has been trained, **cgtc_run_cst()** performs the trained constant intensity GTC method. See **cgtc_train()** for details on training for constant intensity GTC.

If the *trn_meth* component of the **cgtc_rt_trn_params** structure is *not* set to *CGTC_TRN_CST*, **cgtc_run_cst()** performs the non-trained constant intensity GTC method. See **cgtc_train()** for details on training for constant intensity GTC. In this method, the detection of defects for a uniform color sample is based on the run-time computation of the mean and standard deviation of the image. Consequently, unless the image is uniformly lit, any areas of the image that deviate from the mean value by more than the specified threshold will be seen as defects.

The detection of defects for a uniform color sample is based on the run-time computation of the mean and standard deviation of the image. Consequently, unless the image is uniformly lit, any areas of the image that deviate from the mean value by more than the specified threshold will be seen as defects.

img: pointer to the image to process.

dm: pointer to a defect module.

rp: pointer to the run-time parameters.

res: pointer to a GTC results structure. If *res* is NULL, **cgtc_run_cst()** allocates the GTC results structure, and you are responsible for freeing it using **cgtc_delete_results()**. If you allocate the GTC results structure using **cu_alloc()**, the structure must be zeroed out using **cu_clear** before the first time it is called.

gtc_blob: pointer to the blob analysis parameters. See function **cgtc_blob()**.

msigma: the standard deviation multiplier in tens of units. This quantity specifies the number of standard deviation sigma units necessary for a pixel to be classified as a defect. The effective threshold is the mean grey value of the image ± *msigma*/10.

In the trained constant intensity GTC method, the mean and standard deviation are calculated based on the trained image; in the non-trained constant intensity GTC method, the mean and standard deviation are calculated based on the test image.

cgtc_run_cst() returns a pointer to a GTC results structure **cgtc_results**.

ENUMERATION: **cgtc_run_meth**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef enum {
   CGTC_RUN_ST,
   CGTC_RUN_TI,
   CGTC_RUN_EITHER,
   CGTC_RUN_CST
}  cgtc_run_meth;
```

NOTES: **cgtc_run_meth** specifies the run-time inspection method for which the module is trained. If you are running inspection with grey-scale morphological filtering, you should select *CGTC_RUN_TI* or *CGTC_RUN_EITHER*. (3000/4000/5000 only.)

CGTC_RUN_ST: single threshold method. During training, the standard deviation and threshold images are not calculated, but a threshold table is calculated.

CGTC_RUN_TI: threshold image method. During training, the standard deviation and threshold images are calculated, and the threshold table is not calculated.

CGTC_RUN_EITHER: train for both types of inspection. The standard deviation image, threshold image, and threshold table are all calculated.

CGTC_RUN_CST: Train for constant intensity GTC method. This value is set automatically by the training function **cgtc_train()** and signifies the training state of the defect module. You should not set **cgtc_run_meth** to *CGTC_RUN_CST* directly, since this will signify a false trained state to **cgtc_run_cst()**. See **cgtc_run_cst()** for details.

STRUCTURE: **cgtc_run_params**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   short
      adx, ady,
      thr_st,
      thr_cst,
      limit,
      max_area;
   int display;
   cgtc_insp_opts insp_meth;
   void (*norm_func)();
   int blob_flags;
   cse_params *sp;
   short *thresh_tab;
   int res_form;
   int reserved[2];
}  cgtc_run_params;
```

NOTES: **cgtc_run_params** contains the run-time parameters for inspection. *Note*: The default run parameters are pointed to by *dflt_rp* (see the file *cgtc_glb.v*). (3000/4000/5000 only.)

adx, *ady:* the alignment deviation (search range) for the search model; the total search range is larger than the search model *cse_model* by *adx*, *ady*.

thr_st: the global threshold value (in grey-scale units) to be used in the thresholding step *only* when the single thresholding method is selected. In single thresholding method only one value is used to threshold the test image. You specify the threshold value using the function **cgtc_set_st_thresh()**.

thr_cst: the threshold value to be used *only* when constant intensity GTC is selected. *thr_cst* specifies the threshold value in tens of sigmas, where sigmas represent the standard deviation of the run-time image.

limit: the limit at which to stop counting defect pixels.

max_area: the maximum blob area for defect pixels. If the biggest blob has an area greater than *max_area*, the blob computation is aborted and no blob results are reported.

display: flag which controls the amount of information displayed and reported during run time. *Note*: The first three flags, *CGTC_DSP_REG*, *CGTC_DSP_COUNT*, and *CGTC_DSP_TIMES*, are for use by the **cgtc_*** code exclusively; all other flags are used by the template code to control the display of images and graphics.

CGTC_DSP_REG: print alignment information (training time)

CGTC_DSP_COUNT: print training sample counts

CGTC_DSP_TIMES: print run times

CGTC_DSP_OFF: no display

CGTC_DSP_NORM: normalized image

CGTC_DSP_DIFF: difference image

CGTC_DSP_ERR: error image (thresholded difference)

CGTC_DSP_FILTER: filtered error image (show defects after morphology)

CGTC_DSP_TMPL: template (training time)

CGTC_DSP_SD: standard deviation image (training time)

CGTC_DSP_GRAPH: graph of contrast normalization function

CGTC_DSP_BLOB: show blob results on screen

CGTC_DSP_SOB: display Sobel magnitude image (training time)

CGTC_DSP_REGR: print linear regression results (training time)

CGTC_DSP_RUN: all run time displays; this is a combination of the following flags: *CGTC_DSP_NORM*, *CGTC_DSP_DIFF*, *CGTC_DSP_ERR*, *CGTC_DSP_FILTER*, *CGTC_DSP_GRAPH*, *CGTC_DSP_REG*, *CGTC_DSP_TIMES*

CGTC_DSP_TRN: all training time displays; this is a combination of the following flags: *CGTC_DSP_TMPL*, *CGTC_DSP_SD*, *CGTC_DSP_REG*, *CGTC_DSP_COUNT*

CGTC_DSP_ALL: display/print everything

insp_meth: enumeration specifying the inspection method to use:

```
typedef enum {
   CGTC_INSP_ST,
   CGTC_INSP_TI,
   CGTC_INSP_CST,
   CGTC_INSP_GREY
} cgtc_insp_opts;
```

CGTC_INSP_ST: perform the inspection using a single value *thr_st* for the thresholding step (the single threshold method). You specify the threshold value using the function **cgtc_set_st_thresh()**.

CGTC_INSP_TI: perform the inspection using an image for the thresholding step (the threshold image method). The threshold image is derived from training; the function **cgtc_set_ti_thresh()** specifies the linear mapping of the standard deviation into the threshold image.

The threshold image is pointed to by the *timg* component of the **cgtc_tmpl_data** structure

CGTC_INSP_CST: perform constant intensity GTC

CGTC_INSP_GREY: perform inspection with grey-scale morphological filtering

norm_func: pointer to the method used to calculate the mapping in the normalization step. The normalization mapping functions available are: **cgtc_norm_ident()**, **cgtc_norm_tail()**, **cgtc_norm_mean_sd()**, and **cgtc_norm_hist()**.

Each of the normalization functions has the format described below. **cgtc_norm_tail()** is described as an example; its arguments are the same as those for the other normalization functions. You can write your own normalization functions, but you must use the following format.

```
void cgtc_norm_tail (dm, src, td, rp)
   cgtc_module *dm;
   cip_buffer *src;
   cgtc_tmpl_data *td;
   cgtc_run_params *rp;
```

dm: pointer to the defect module. Note that the pixel mapping for normalization must be stored in *dm->map*.

src: pointer to the source image to be normalized (typically the test image).

td: pointer to the template data structure.

rp: pointer to the run-time parameters.

Note: The pixel mapping for normalization must be stored in *dm->map*.

If you set *norm_func* to NULL, the normalization function defaults to **cgtc_norm_ident()**.

The name of the normalization function is reported in the GTC results structure if the *res_form* component of the **cgtc_run_params** structure is *CGTC_RES_NFUN*.

blob_flags: the blob computation method, i.e., how blob statistics are computed, according to the following flags. *Note*: The blob computation represents one-third of the time required for inspection.

CGTC_BLOB_OFF: No blob analysis is performed

CGTC_BLOB_ZERO: Area of each blob is computed

CGTC_BLOB_ONE: up to 1st moment of each blob is computed

CGTC_BLOB_TWO: up to 2nd moment of each blob is computed (this is the default blob analysis function)

CGTC_BLOB_DIR: The grey value blobs are computed directly from the grey value difference image *dimg*. This method is less accurate than *CGTC_BLOB_DIF*, but it is faster.

CGTC_BLOB_DIF: The grey value blobs are computed from the grey value error image, i.e., the difference between the threshold image *timg* and the absolute value of the difference image *dimg*. Because only the binary error image is available, the grey value error image must be calculated by **cip_arith()**, due to the computation of the grey value error image. This method is more accurate than *CGTC_BLOB_DIR*, but it is slower.

CGTC_BLOB_DFLT: Default blob computation method; it computes up to the 2nd moment of the blob, using the difference image

You must specify the level of the blob analysis method by selecting one of the following: *CGTC_BLOB_ZERO*, *CGTC_BLOB_ONE*, or *CGTC_BLOB_TWO*; you must also specify the blob computation method by selecting either *CGTC_BLOB_DIR* or *CGTC_BLOB_DIF*. If you do not specify a blob computation method, the GTC software executes *CGTC_BLOB_DIR* by default.

sp: pointer to the search parameters for alignment. *Note*: *cse_params* is described in the *Searching* chapter of the *Vision Tools* manual.

thresh_tab: pointer to the threshold table used by the single threshold method. It is created by calling **cgtc_set_st_thresh()**.

res_form: flag indicating which optional results appear in the results structure:

CGTC_RES_OFF: no optional results are included in the result structure

CGTC_RES_MFUN: report the name of the morphology function in the result structure, as specified by *morph_func* in the defect module.

CGTC_RES_NFUN: report normalization function name in result structure

CGTC_RES_SR: a copy of search results is included in the result structure

CGTC_RES_TD: include a link to the template data (*tmpl_data*) in the result structure

CGTC_RES_GBUF: a link to the windows *dimg*, *eimg*, and *fimg* in the **cgtc_module** structure is included in the result structure

CGTC_RES_STD: default setup; catenation of the following: *CGTC_RES_SR*, *CGTC_RES_TD*, *CGTC_RES_GBUF*

CGTC_RES_ALL: all optional results are included in the result structure

reserved: reserved for future use.

FUNCTION: **cgtc_set_morph()**

PURPOSE: Changes the specified morphology function in the defect module (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_set_morph(dm, morph_func)
   gtc_module *dm;
   void (*morph_func)();
```

NOTES: You must specify the morphological operation (opening, closing, dilation, or erosion) by setting the function pointer *morph_func* of the function **cgtc_set_morph()** to point to one of the morphology functions.

Note: **cgtc_set_morph** sets the function pointer *morph_func* in the structure **cgtc_module**, and recomputes *fmask*. Because changing the morphology function requires both of these tasks, this function provides the *only* legal way to change the name of the morphology function.

The available morphology functions are listed in the *GTC* chapter of the *Vision Tools* manual. If you want to disable the morphology operation, specify the identity function **cgtc_morph_ident()**.

dm: the defect module that contains the morphology function.

morph_func: pointer to the name of the morphological function.

FUNCTION: **cgtc_set_st_thresh()**

PURPOSE: Recomputes the global threshold table for the single threshold method (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_set_st_thresh (rp, thresh)
   cgtc_run_params *rp;
   int thresh;
```

NOTES: **cgtc_set_st_thresh()** does *not* set or change the value of *thr_st* in the structure **cgtc_run_params**. If you do not call this function the threshold table will be derived from the value of *thr_st*.

rp: pointer to the run-time parameters structure **cgtc_run_params** which contain the threshold table *thresh_tab*.

thresh: the threshold value. **cgtc_set_st_thresh()** calculates the threshold table based on the threshold value you provide.

FUNCTION: **cgtc_set_ti_thresh()**

PURPOSE: Maps the standard deviation image into the threshold image and transfers the mask image onto the threshold image (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_set_ti_thresh (dm, thr_cst, thr_slp, index)
   cgtc_module *dm;
   int thr_cst;
   double thr_slp;
   int index;
```

NOTES: **cgtc_set_ti_thresh()** computes an 8-bit threshold image if the defect module is trained using the VC-2 coprocessor (and grey-scale morphology is specified); otherwise the threshold image is a 7-bit image.

dm: pointer to the defect module.

thr_cst: the threshold constant.

thr_slp: the threshold slope.

index: an integer specifying which subpixel bin of *sdimg* to map. -1 specifies all bins.

FUNCTION: **cgtc_setup_graphics_luts()**

PURPOSE: Sets up the graphics lookup tables for GTC (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:
```
void cgtc_setup_graphics_luts ()
```

NOTES: **cgtc_setup_graphics_luts()** sets up the graphics lookup tables for GTC. You must call it *once only* from your application program to enable the lookup tables used for the graphics displays.

FUNCTION: **cgtc_sobel()**

PURPOSE: Computes the Sobel magnitude of a source image and returns the Sobel magnitude image (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *None*

USE:

```
cip_buffer *cgtc_sobel (src, mag, edge_p)
   cip_buffer *src, *mag;
   cip_edge_params *edge_p;
```

NOTES: Sobel magnitude is discussed in the *Edge Detection Tool* chapter of the *Image Processing* manual.

src: pointer to the source image.

mag: pointer to the Sobel magnitude image. If *mag* is NULL, this function allocates a Sobel magnitude image, and you are responsible for deallocating it.

edge_p: pointer to the edge parameters structure. For a description of the **cip_edge_params** structure, see the *Edge Detection Tool* chapter in the *Image Processing* manual.

FUNCTION: **cgtc_swap()**

PURPOSE: Swaps the images contained in a GTC defect module to and from a SCSI disk (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_swap (dm, dbname, swap_p, index)
   cgtc_module *dm;
   char *dbname;
   cgtc_swap_params *swap_p;
   int index;
```

NOTES: **cgtc_swap()** swaps the images contained in a GTC defect module to and from a SCSI disk. Images represent more than 99% of the space occupied by a defect module, and it might be necessary to clear them while leaving the defect module resident for later use. The swap file is implemented as a database file on a SCSI disk. In swap-out mode *CGTC_SWP_OUT*, the database is opened in write-only mode and the contents of the swap file are overwritten. In particular, all the images that are going to be swapped out should be written in a single operation. With swap-in mode *CGTC_SWP_IN*, the database file is opened in read-only mode, and selected images can be swapped in. During a swap-out, the images can be optionally deleted form the defect module.

dm: the resident defect module.

dbname: pointer to the name of the swap file.

swap_p: pointer to the input/output parameters used during swapping.

index: the index of the bin to transfer. The index indicates which bin is transferred in or out. A value of -1 indicates that all the bins of the selected image will be transferred.

cgtc_swap() throws the following errors:

- *CGTC_ERR_BADIDX* is thrown is an out-of-bound index is specified.
- *CGTC_ERR_SWP_NOIMG* is thrown when a image that is not in the swap database is requested to be swapped in.
- *CGTC_ERR_SWP_WRNGSIZE* is thrown if the size (width, height, depth) and position (*ix0,iy0*) of the stored image in the swap database and the corresponding image in the module do not agree during a swap in operation.
- *CGTC_ERR_SWP_WRNGIMG* is thrown when the result (header) and following image records do not agree.

STRUCTURE: **cgtc_swap_params**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct{
   int itype;
   bool delete;
   cgtc_swap_xfer xfer;
}  cgtc_swap_params;
```

NOTES: **cgtc_swap_params** contains the input/output parameters used during swapping. These parameters specify which images to transfer, the direction of transfer and the deletion of the image inside the defect module. (3000/4000/5000 only.)

itype: an integer specifying which image to swap, according to the following flags:

CGTC_SWP_TMPL: template image

CGTC_SWP_SDIMG: standard deviation image

CGTC_SWP_TIMG: threshold image

CGTC_SWP_SUMS: sum image

CGTC_SWP_MASK: mask image

delete: a boolean that indicates whether the image is deleted in the module. It is only meaningful in the mode *CGTC_SWP_OUT* or *CGTC_SWP_NO*.

xfer: an enumeration controlling which swap operation to perform:

```
typedef enum {
   CGTC_SWP_IN,
   CGTC_SWP_OUT,
   CGTC_SWP_NO
} cgtc_swap_xfer;
```

CGTC_SWP_IN: image is swapped in (disk to module)

CGTC_SWP_OUT: image is swapped out (module to disk)

CGTC_SWP_NO: no disk transfer, only delete operation

STRUCTURE: **cgtc_tmpl_data**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   cip_buffer
      *tmpl,
      *timg,
      *sdimg,
      *sums;
   short n;
   int *hist;
   unsigned char lt, rt;
   float mean, sd;
   int tmean, tmed;
   int reserved[2];
}  cgtc_tmpl_data;
```

NOTES: **cgtc_tmpl_data** contains the template data for one subpixel bin (there is a separate template data structure for each subpixel bin). **cgtc_tmpl_data** contains pointers to the statistical images (for example, template, standard deviation, etc.) plus other variables for statistical analysis. The training software sets the values of components in **cgtc_tmpl_data**, which contains all the statistical information required for inspection for each subpixel bin. (3000/4000/5000 only.)

tmpl: pointer to the template image.

timg: pointer to the threshold image (an 8-bit global **cip_buffer** the size of the frame grabber).

sdimg: pointer to the standard deviation image (an 8-bit global **cip_buffer** the size of the frame grabber).

sums: pointer to the accumulator image used for training.

n: number of sample images acquired.

hist: pointer to the histogram of the template image.

lt, rt: the left and right tails, respectively, of the template image.

mean: mean grey value of the template image.

sd: standard deviation of the template image.

tmean, tmed: the integer mean and median, respectively, of the threshold image.

reserved: reserved for future use.

FUNCTION: **cgtc_train()**

PURPOSE: Calculates all the statistical images based on the sample images acquired (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
void cgtc_train (img, sr, dm, rp, trnp, index)
   cip_buffer *img;
   cse_results *sr;
   cgtc_module *dm;
   cgtc_run_params *rp;
   cgtc_rt_trn_params *trnp;
   int index;
```

NOTES: The computed threshold images will be 8-bit images if the defect module is trained using the VC-2 coprocessor, or 7-bit images using the VC-1 coprocessor. The upper 16 bits of the *id* component of the **cgtc_module** structure is set accordingly.

img: pointer to the whole-pixel aligned sample image to process.

sr: pointer to the search results, which are used to align *img*.

dm: pointer to the defect module.

rp: pointer to the run-time inspection parameters.

trnp: pointer to the parameters used in the training of sample images. When you call **cgtc_train()**, it performs the training operation specified by the setting of the training method *trn_meth* of the **cgtc_rt_trn_params** structure. If *trn_meth* is set to *CGTC_TRN_CST*, **cgtc_train()** sets the *meth* component of the **cgtc_trn_params** structure to *CGTC_RUN_CST* and trains the image *img* for constant GTC. You should not set *meth* to *CGTC_RUN_CST* directly; doing so may cause the software to indicate a false trained state.

index: the index of the subpixel bin to use; -1 specifies all bins.

cgtc_train() throws the following errors:

- *CGTC_ERR_NOTTRN* if computation of *statistics* mode has been requested, and the *sums* image is NULL or number of samples = 0
- *CGTC_ERR_BADIDX* if the argument *index* is less than zero, or greater than the number of bins defined in the defect module *dm*
- *CGTC_ERR_NOSOBIMG* if you do not run the Sobel operation *CGTC_TRN_SOB* before running the Sobel mapping *CGTC_TRN_MPSOB*

FUNCTION: **cgtc_transmit_module()**

PURPOSE: Converts a GTC defect module from its Cognex internal format to a byte stream format for transfer to nonvolatile memory, to a serial or parallel port, or to other hardware (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtcxfer.h*

USE:

```
int cgtc_transmit_module(dm, name, xfer_params, output_fn,
      output_params, gtc_save)
   cgtc_module *dm;
   char *name;
   cio_xfer_params *xfer_params;
   c_function output_fn;
   char *output_params;
   int gtc_save;
```

NOTES: **cgtc_transmit_module()** is independent of any hardware device. You select the device by passing a pointer to an output function.

dm: pointer to the defect module to transfer.

name: pointer to the host's name for the defect module, which is passed to the I/O function during initialization. This is usually used as a filename but may be NULL. For a complete description of how *name* is processed, see the *I/O* chapter of the *System Software* manual.

xfer_params: pointer to the transfer parameters. These parameters are specified by the **cio_xfer_params** data structure and define such things as compression style, data transfer formats, and ASCII line termination. Refer to the description of the **cio_xfer_params** data structure in the *I/O* chapter of the *System Software* manual for detailed information on transfer parameter settings.

If you specify NULL for *xfer_params*, the following defaults are used:

```
cio_xfer_params.style = CIO_XFER_STYLE_PACKED
cio_xfer_params.prep = CIO_XFER_PREP_ASCII
cio_xfer_params.ascii.chars_per_line = 78
cio_xfer_params.ascii.ending = CIO_XFER_ENDING_LF
```

output_fn: pointer to a function to send the defect module to the hardware. You can write your own output function, or use one of the Cognex output functions. Refer to the *I/O* chapter in the manual, *System Software*, for more information about the Cognex output functions.

If you specify NULL for the *output_fn* argument, the system returns a byte count without actually transmitting any data.

Note: Use caution when specifying NULL for the *output_fn* argument. If you are using an output function that transmits more characters than

it receives (this is not the case with the Cognex output functions), then the byte count returned when using NULL for the *output_fn* argument will be incorrect. It is best to let the output function return the byte count by specifying the flag *CIO_XFER_GEN_FLAG_COUNT_ONLY* in the I/O parameters structure.

output_params: pointer to parameters to be sent to the output function. If you are using a Cognex output function, then the output parameters are specified by a parameters structure specific to the type of transfer taking place. Refer to the *I/O* chapter of the *System Software* manual for detailed information on I/O parameters structures.

If you are using your own output function, you can pass data to it with the *output_params* pointer.

gtc_save: flag indicating which components of the module are saved.

CGTC_S_NO: do not save histogram or images.

CGTC_S_MASK: save mask image *cgtc_module->mask*.

CGTC_S_TMPL: save template image *cgtc_tmpl_data->tmpl*.

CGTC_S_TIMG: save the threshold image *cgtc_tmpl_data->timg*.

CGTC_S_SDIMG: save standard deviation image *cgtc_tmpl_data->sdimg*.

CGTC_S_SUMS: save accumulator image *cgtc_tmpl_data->sums*.

CGTC_S_HIST: save histogram *cgtc_tmpl_data->hist*.

CGTC_S_STD: standard (default) save; this is a combination of the following flags: *CGTC_S_TMPL*, *CGTC_S_SDIMG*, *CGTC_S_HIST*.

CGTC_S_ALL: save everything.

cgtc_transmit_module() throws the error *CGHT_ERR_BADARG* if any argument is not valid. It throws the error *CGHT_ERR_PARAMS* if a field of a parameters structure is not valid.

STRUCTURE: **cgtc_trn_params**

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

DEFINITION:

```
typedef struct {
   short
      spx, spy,
      bits;
   short thr_cst;
   float thr_slp;
   float left_tail, right_tail;
   cgtc_run_meth meth;
   short xavg, yavg;
   bool all_bin;
   short num_cse_mod;
   int reserved[2];
}  cgtc_trn_params;
```

NOTES: **cgtc_trn_params** contains the training parameters. (3000/4000/5000 only.)

spx, *spy:* the x and y subpixel resolutions (the number of horizontal and vertical subpixel bins), respectively (e.g. a value of *spx* = 2 and *spy* = 1 specifies a 2x1 subpixel resolution).

bits: number of most significant bits in each pixel to use for statistical training.

thr_cst: the constant component of the threshold image. The constant component is *b* in the linear equation $y = ax + b$; also, *x* is the standard deviation image and *y* is the resulting threshold image. Note that *thr_cst* is used only for the threshold image method.

thr_slp: the sigma (slope) of the mapping from the standard deviation to the threshold (the sigma is the slope *a* in the linear equation $y = ax + b$). Note that *thr_slp* is used only for the threshold image method.

left_tail, *right_tail:* the left and right tails of the histogram (for optional use in the normalization step).

meth: the run-time method for which the module is trained.

xavg, *yavg:* the horizontal and vertical averaging factors of the sample image.

all_bin: a boolean which specifies whether the first sample fills all the subpixel bins until a valid sample is collected. *all_bin* is used in cases in which it is necessary to train a module with a single sample before taking a fully trained multiple bin sample.

num_cse_mod: number of search model (**cse_model**) structures available (equivalent to the size of the search model array *model* in **cgtc_module**).

reserved: reserved for future use.

FUNCTION: **cgtc_window_module()**

PURPOSE: Creates a window of a specified defect module (3000/4000/5000 only).

REFERENCE: Vision Tools: Golden Template Comparison

INCLUDE FILE: *cgtc.h*

USE:

```
cgtc_module *cgtc_window_module (dm, wdm, xorig, yorig,
      wd, ht)
   cgtc_module
      *dm, *wdm;
   int
      xorig, yorig,
      wd, ht;
```

NOTES: **cgtc_window_module()** creates a *window module* from a specified defect module called the *parent module*. All the images in the window module are windows of corresponding images or global images in the parent module, and the window statistics are recomputed. The search model in the window mod-*dm* points to the parent defect module.

dm: pointer to the parent defect module.

wdm: pointer to the window defect module. If *wdm* is NULL, **cgtc_window_module()** creates a new window defect module.

xorig and *yorig:* the coordinates of the upper left corner of the windows relative to the images in the parent module.

wd and *ht:* the width and height of the window.

STRUCTURE: **chlt_data**

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

DEFINITION:

```
typedef struct {
   chlt_params params;
   cip_edge_data *edge_p;
   cip_edge_results edge_results;
   short *sin_cos_p;
   char *ang_lim_tbl_p;
} chlt_data;
```

NOTES: **chlt_data** contains the tables used by the Line Finder, along with a copy of the **chlt_params** structure used to build the tables. It is built by the function **chlt_make()**.

params: a copy of the parameters used to create this **chlt_data** structure. Using this nested structure, you can change the following parameters at runtime, without re-invoking **chlt_make()**: *num_results, resolution, flags, report, line_thr, mag_thr, dens_thr, length_thr, input, dsp_image,* and *color.*

edge_p: points to the **cip_edge_data** structure that was generated using the **cip_edge_params** structure passed to **chlt_make().**

edge_results: a results record filled by the edge detection function **cip_sobel()**, which is invoked by **chlt_make()**. You can use *edge_results* to access the magnitude and angle images used by the Line Finder.

sin_cos_p: points to a table that maps an angle θto the sine and cosine of the angle.

ang_lim_tbl_p: points to a table that maps an angle θ to TRUE if **chlt_run()** should locate lines of angle θand FALSE otherwise. This table is built if the *CHLT_FLAG_LIMIT_ANG* flag is set in the **chlt_params** data structure. It is created by using either the *ang_lim_fnp* and *lim_params_p* fields or the *base_ang* and *ang_tol* fields of the **chlt_params** structure.

FUNCTION: **chlt_draw()**

PURPOSE: **chlt_draw()** draws a line, as described in a **chlt_results** structure, to the display image pointed to from the input **chlt_params** structure.

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

USE:

```
void chlt_draw(params_p,results_p)
   chlt_params *params_p;
   chlt_results *results_p;
```

NOTES: If the result was computed in normal mode, the line is drawn to the length stored in the **chlt_results** structure. If the result was computed in quick mode, the line is drawn to a constant length (approximately 80 pixels).

params_p: points to the **chlt_params** structure containing the pointer to the display image.

results_p: points to the **chlt_results** structure containing the line.

This function throws the following errors:

- *CGEN_ERR__BADARG,* if either argument is NULL
- *CHLT_ERR_BAD_DISPLAY,* if *params_p->dsp_image* is NULL

FUNCTION: **chlt_free()**

PURPOSE: **chlt_free()** frees all memory used by the input **chlt_data** structure. This includes any tables pointed to within the structure.

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

DEFINITION:

```
void chlt_free(data_p)
   chlt_data *data_p;
```

NOTES: *data_p:* points to the **chlt_data** structure to free.

chlt_free() throws the error *CGEN_ERR__BADARG* if *data_p* is NULL.

ENUMERATION: **chlt_input**

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

DEFINITION:

```
typedef enum {
   CHLT_IN_IMAGE,
   CHLT_IN_EDGE_MA,
   CHLT_IN_EDGE_XY,
   CHLT_IN_PEAK
} chlt_input;
```

NOTES: **chlt_input** specifies the types of images that can be used as input to the Hough line function, **chlt_run()**:

CHLT_IN_IMAGE: a standard input image, not an edge detection magnitude image

CHLT_IN_EDGE_MA: an edge detection magnitude and angle images

CHLT_IN_EDGE_XY: not implemented

CHLT_IN_PEAK: not implemented

FUNCTION: **chlt_make()**

PURPOSE: **chlt_make()** generates a **chlt_data** structure for a set of parameters.

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

USE:

```
chlt_data *chlt_make(data_p, params_p)
   chlt_data *data_p;
   chlt_params *params_p;
```

NOTES: *data_p:* points to the output **chlt_data** structure. If this pointer is NULL, **chlt_make()** allocates and fills this structure and returns its address. If *data_p* points to an existing **chlt_data** structure, **chlt_make()** operates in one of the following two ways:

params_p: points to an input **chlt_params** structure. This structure contains data controlling the creation of the edge detection tables used by **chlt_run()** along with other input data. This field can be NULL only if *data_p* (above) is non-NULL.

If *params_p* is non-NULL, **chlt_make()** rebuilds the **chlt_data** structure, using the parameters pointed to by *params_p*.

If *params_p* is NULL, **chlt_make()** rebuilds the **chlt_data** structure, using the parameters stored in *data_p->params*, which is of type **chlt_params**.

chlt_make() throws the following errors:

- *CHLT_ERR_BAD_ANGLE*, if *params_p->base_ang* is less than zero or greater than 360
- *CHLT_ERR_BAD_TOL*, if *params_p->angle_tol* is less than zero or greater than 360
- *CHLT_ERR_NO_180*, if *params_p->edge.flags* contains the setting *CIP_EDGE_FLAG_180*
- *CHLT_ERR_XY_IN_NOTIMP* if *params_p->input* is *CHLT_IN_EDGE_XY*
- *CHLT_ERR_PEAK_IN_NOTIMP* if *params_p->input* is *CHLT_IN_PEAK*
- *CGEN_ERR_BADARG* if both *data_p* and *params_p* are NULL

STRUCTURE: **chlt_params**

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

DEFINITION:

```
typedef struct {
   cip_edge_params edge;
   short
      dis_radius, num_results,
      resolution, flags, report;
   float
      mag_thr, line_thr, dens_thr;
   short length_thr;
   chlt_input input;
   cip_buffer *dsp_image;
   short
      ang_grp_tol, color, base_ang, ang_lim_tol;
   bool (*ang_lim_fnp)();
   char *lim_params_p;
} chlt_params;
```

NOTES: **chlt_params** contains the input to the function **chlt_make()**, which generates the tables used by the runtime function **chlt_run()**.

edge: an initialized **cip_edge_params** data structure. This structure is used by the function **chlt_make()** to create the tables used during the edge detecting phase of the Hough line algorithm.

dis_radius: the radius of Hough space in pixels. *dis_radius* is the largest possible distance to a line from the center of the image. It applies to the reduced resolution image. Therefore, the farthest line that the Line Finder can resolve is at distance *dis_radius* multiplied by *resolution*.

num_results: the maximum number of lines to find in the input image.

resolution: the spatial averaging rate for the input image. You should specify a reduced resolution only if *chlt_params.input* is *CHLT_IN_IMAGE*. By reducing resolution you can significantly improve the speed of the Line Finder at the risk of losing weak edges. Also, at reduced resolution the reported positions of lines are less accurate and parallel lines that are close to one another may be merged. To specify normal resolution, set this field to 1.

flags: contains input switches to **chlt_make()**. The following flags are defined:

- *CHLT_FLAG_NO_STATS:* operate in *quick mode*. If this flag is not set, the Line Finder operates in *normal mode*, the default.
- *CHLT_FLAG_LIMIT_ANG:* find lines only within one or more specified angle ranges. These ranges are provided in the *base_ang*, *angle_lim_tol*, *ang_lim_fnp*, and *lim_params* fields of this structure.

report: a flag word that controls the information that **chlt_run ()** supplies about each line that it locates. The information includes data about each line along with timing information. The following flags are defined:

- *CHLT_REPORT_POINT:* print the Cartesian coordinates and angle of every edge point in the input image that is entered in Hough space. The coordinates are given relative to the central pixel of the image. The angle is provided in decimal and binary form.
- *CHLT_REPORT_RSLT:* print the Hough space coordinates of each line. If the Line Finder is in normal mode, also print the length and density of each line segment.
- *CHLT_REPORT_PLOT:* plot, in the display image, all edge points entered in Hough space.
- *CHLT_REPORT_TIME:* print the total amount of time that it takes to locate all lines. This report is broken into the following categories:
  - time to spatially average the input image
  - time to run edge detection on the input image
  - time to enter edge points to Hough space
  - time to calculate and fill results records
  - other time
- *CHLT_REPORT_PRINT:* print the contents of the **chlt_results** record for each line.
- *CHLT_REPORT_DRAW:* draw each line in the display image.

mag_thr: the minimum edge magnitude that a pixel must have to be entered in Hough space.

line_thr: the lowest value at which an entry in Hough space should be considered a line. There is no practical maximum value for a score in Hough space. Also, the range of scores and the required attributes of lines differs from application to application. You should set this field to 20 and run the Line Finder on several images that are typical for your application, and then adjust *line_thr* depending on the results of your experiments.

dens_thr: the lowest density that a line candidate can have and be considered a line. As with *line_thr*, density threshold is very application-specific. You should set this field to 1.5, run the Line Finder on several images that are typical for your application, and then adjust *dens_thr* depending on the results of your experiments. This field is used only if the Line Finder is operating in normal mode.

length_thr: the minimal acceptable length of a line, in pixels. This field is used only if the Line Finder is operating in normal mode.

input: describes the input images that will be passed to **chlt_run()**, as enumerated by the type **chlt_input**.

dsp_img_p: points to a display image in which points and lines are plotted by **chlt_run()**. If this field is NULL and *input*, below, is *CHLT_IN_IMAGE*, **chlt_run()** uses the input image as the display image and places a pointer to it in this field.

angle_grp_tol: reserved for future development.

color: the pixel value to use for plotting points in the display image.

base_ang: used only if *CHLT_FLAG_LIMIT_ANG* is set in *chlt_params.flags* and if you want to find lines within a single angle range. This field supplies a base angle that is used with *ang_tol*, below, to calculate the angle range within which **chlt_run()** should find lines. This field must be in the range 0-360.

ang_lim_tol: used only if *CHLT_FLAG_LIMIT_ANG* is set in *chlt_params.flags* (above) and if you want to find lines within a single angle range. This field supplies the positive and negative angle tolerance that is used with *base_ang* (above) to calculate the angle range within which **chlt_run()** should find lines. This field must be in the range [0 - 360].

base_ang and ang_tol are combined to compute the angle range. If *base_ang* is θ and *ang_tol* is t, the angle range is (θ- t) to (θ + t), with the following adjustments:

If (θ - t) is negative, it is changed to (360+θ- t).

If (θ + t) is greater than or equal to 360, it is changed to (θ +t- 360).

ang_lim_fnp: used only if *CHLT_FLAG_LIMIT_ANG* is set in *flags*. It is a pointer to a boolean function that takes, as input, an angle θ, in decimal form, and returns TRUE if you want **chlt_run()** to find lines of angle θ. If this field is non-NULL, **chlt_make ()** ignores the previous two fields, *base_ang and ang_tol*.

This field lets you specify one or more angles, not necessarily within a single range, at which **chlt_run()** should locate lines. You should use it (and the field *lim_params_p*, below) if you want to specify more than one angle range for **chlt_run().**

The function pointed to by *ang_lim_fnp* must take an angle and a pointer as input. The next field, *lim_params_p*, is provided to pass your function the pointer.

lim_params_p: a pointer that is available to you to pass data to the function pointed to by *ang_lim_fnp* (above).

FUNCTION: **chlt_print()**

PURPOSE: **chlt_print()** prints the contents of a **chlt_results** structure.

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

USE:

```
void chlt_print(params_p,results_p)
   chlt_params *params_p;
   chlt_results *results_p;
```

params_p: points to the **chlt_params** structure that was used to compute the results.

results_p: points to the **chlt_results** structure containing the data. If this field is NULL, **chlt_print()** prints a row of headings that label each column of output.

chlt_print() throws the error *CGEN_ERR_BADARG* if *params_p* is NULL.

STRUCTURE: **chlt_results**

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

DEFINITION:

```
typedef struct {
   short
      cmx,cmy,
      theta,
      cmc,cmd;
   float
      polarity,score,est_length,density;
   chlt_results *parallel_p;
} chlt_results;
```

NOTES: **chlt_results** contains the data for a single line found by the Hough line function, **chlt_run()**.

cmx, cmy: the image buffer coordinates of a point along the line. If the Line Finder is in quick mode, these are the coordinates of the point along the line nearest to the center of the image. If the line finder is in normal mode, these are the coordinates of the center of mass of the line.

theta: the angle of the line, in Cartesian coordinates. It is an integer, expressed in binary form, that is greater than or equal to zero and less than 180. If *polarity* (below) is negative, add 180 to *theta* to get an angle in the range [0 - 360].

Note: In the Line Finder, the binary representation of an angle is a fixed point number with the fixed point to the left of the most significant bit and is interpreted as a fraction of 180°. If *n* is the maximum number of bits used to express angle (as specified in the **cip_edge_params** data structure), and *x* is the binary representation of an angle, you can convert *x* to degree notation by applying the following formula:

$$angle\,(x) = \frac{180x}{2^n}$$

cmc: the colinear distance to the center of mass of the line, if the Line Finder is in normal mode; otherwise *cmc* is zero.

cmd: the distance from the center point of the image to the nearest point on the line.

polarity: a number between -1.0 and +1.0 that indicates the proportion of pixels along a line with angles less than 180° to those pixels with angles greater than or equal to 180°. The *polarity* of a line is defined as follows:

$$\frac{L-G}{T}$$

where L is the number of pixels on the line with angles greater than or equal to 0° and less than 180°, G is the number of pixels on the line with angles greater than or equal to 180° and less than 360°, and T is the total number of pixels along the line.

score: the score of the line in Hough space.

est_length: a statistical approximation of the length of the line, in pixels. It is valid only if the Line Finder is in normal mode.

density: the number of pixels contributing to the line divided by the standard deviation of the line. A high density indicates that the *est_length* gives a good approximation of the line's length; a low density indicates that *est_length* may not be accurate. This field is valid only if the Line Finder is in normal mode.

parallel_p: reserved for future development.

FUNCTION: **chlt_run()**

PURPOSE: Finds one or more lines in an input image using the Hough line transform.

REFERENCE: Vision Tools: Line Finder

INCLUDE FILE: *chlt.h*

USE:

```
int chlt_run(image_p,alt_image_p,data_p,results_p)
   cip_buffer *image_p, *alt_image_p;
   chlt_data *data_p;
   chlt_results *results_p;
```

NOTES: The coordinates of each line found, along with other data, are returned in an array of **chlt_results** data structures. The return value is the number of lines found by **chlt_run()**.

image_p: points to an input image. If *data_p->params.input* is *CHLT_IN_IMAGE*, this should be a standard input image. If *data_p->params.input* is *CHLT_IN_EDGE_MA*, this should be an edge magnitude image.

alt_image_p: points to an alternate input image. If *data_p-> params.input* is *CHLT_IN_IMAGE*, this should be NULL. If *data_p->params.input* is *CHLT_IN_EDGE_MA*, this should be an edge angle image.

data_p: points to an initialized **chlt_data** structure.

results_p: points to an array of **chlt_results** structures. **chlt_run()** fills at most *data_p->params.num_results* structures. If you have not allocated enough memory, **chlt_run()** may overwrite other memory.

chlt_run() throws the following errors:

- *CHLT_ERR_BAD_REPORT* if *data_p->params.report* contains an undefined flag
- *CHLT_ERR_BAD_RES* if *data_p->params.resolution* is less than zero
- *CHLT_ERR_BAD_DISPLAY,* if *data_p->params.dsp_img* is NULL and *data_p->params.input* is not *CHLT_IN_IMAGE*
- *CGEN_ERR_BADARG* if *image_p, data_p,* or *results_p* is NULL, or if *data_p->params.input* is *CHLT_IN_EDGE_MA* and *alt_image_p* is NULL

VARIABLE: **chp_alloc_total**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

DEFINITION:

```
int chp_alloc_total;
```

NOTES: **chp_alloc_total** contains the total number of bytes allocated since the system was last booted. It is updated every time **malloc()**, **realloc()**, or **cvc_alloc()** allocates bytes. It includes overhead bytes used by the heap management code as well as the user storage.

chp_alloc_total can overflow and be represented as a negative number.

FUNCTION: **chp_alloca()**

PURPOSE: Allocates temporary storage from the stack.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

USE:

```
char *chp_alloca(size)
   int size;
```

NOTES: **chp_alloca()** returns a pointer to the allocated space. If necessary, **chp_alloca()** calls **cu_expand_stack()**.

size: the number of bytes to allocate.

You must use **chp_alloca()** only inside another function. The storage allocated is temporary and is freed automatically when the calling function returns. **chp_alloca()** is the preferred way of reserving large amounts of storage on the stack (amounts greater than 1Kbyte) because it checks the stack to make sure that enough space is available. If enough space is not available it calls **cu_expand_stack()**.

Important:

Calls to **chp_alloca()** must be embedded only in simple expressions such as:

```
void some_function(){
   int s;
      .
      .
      s = chp_alloca(MAX_SIZE);
      .
      .
   return;
}
```

If **chp_alloca()** is called in more complex expressions, the stack may be corrupted, leading to unpredictable behavior.

In the case where **chp_alloca()** calls **cu_expand_stack()** to reserve additional stack space, you may wish to call **cu_expand_stack()** to reduce the stack size again. If you wish to have complete control over how much stack space is reserved, you should call **cu_expand_stack()** before calling **chp_alloca()**.

CU_ERR_STACK is thrown if the stack cannot be made large enough to satisfy the request.

VARIABLE: **chp_check**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

DEFINITION:

```
int chp_check;
```

NOTES: **chp_check** specifies whether heap consistency checking is to be performed whenever **cvc_alloc()**, **malloc()**, **free()**, or **realloc()** is called. If *chp_check* is nonzero, a consistency checking procedure is performed.

The consistency check is intended for situations where you believe the heap is being corrupted and you want to find out where the corruption is occurring.

Heap consistency checking verifies the internally maintained list of free blocks and checks the allocation count stored before each allocated block. Heap checking can detect most instances of a program modifying memory before the start of a block, or past the end of a block. When an error is detected, the function throws *CHP_ERR_CORRUPT*.

No further heap errors are detected after the error *CHP_ERR_CORRUPT* is thrown. At this point, the operation of the system is unpredictable and you should reboot.

Since consistency checking causes all the heap functions to execute slowly, and many of the vision processor functions and the V compiler use the heap extensively, you should not enable consistency checking unless you suspect an error.

VARIABLE: **chp_free_total**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

DEFINITION:

```
int chp_free_total;
```

NOTES: **chp_free_total** contains the total number of bytes freed since the system was last booted. *chp_free_total* is updated every time **free()** or **realloc()** frees bytes.

chp_free_total can overflow and be represented as a negative number.

FUNCTION: **chp_free_trace()**

PURPOSE: Frees the list of buffers which begin at the specified address.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

USE:
```
void chp_free_trace(header)
   chp_trace_hdr *header;
```

NOTES: *header:* points to the list of transcript buffers to be freed. Since there may be more than one buffer, you cannot free them simply by freeing the first one.

chp_free_trace() throws *CHP_ERR_TRACEON* if you attempt to free a transcript being used by an active heap trace.

VARIABLE: **chp_maxnet**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

DEFINITION:

```
int chp_maxnet;
```

NOTES: **chp_maxnet** contains the maximum difference between *chp_alloc_total* and *chp_free_total* at any one time since the system was last booted.

Note: Although *chp_alloc_total* and *chp_free_total* can overflow and be represented as negative numbers, *chp_maxnet* still will be represented correctly as a positive integer.

ENUMERATION: **chp_print_mode**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

DEFINITION:

```
typedef enum{
   chp_print_full,
   chp_print_unbalanced,
   chp_print_edited
}chp_print_mode;
```

NOTES: **chp_print_mode** specifies the values that the *mode* element of **chp_print_trace()** can have. If *mode* is *chp_print_full*, every heap transaction is reported. If *mode* is *chp_print_unbalanced,* only those heap transactions which were unbalanced are printed; that is, only those in which a block was allocated but not freed, or freed without having been allocated during the time the trace was active. If *mode* is *chp_print_edited*, all the unbalanced transactions except those requested by the trace code itself are printed. Allocations for transcript buffers are by definition unbalanced, since they are never freed until **chp_free_trace()** is called, at which point the transcript can no longer be printed.

The modes *chp_print_unbalanced* and *chp_print_edited* are very similar. An edited transcript is more useful if you are concerned only with which of your functions are performing unbalanced heap operations; an unbalanced transcript accounts for every unbalanced heap transaction.

Although only one heap trace may be active at a time, **chp_print_trace()** can be called any number of times to display the transcript of the ongoing trace's current state.

FUNCTION: **chp_print_stats()**

PURPOSE: Prints the usage statistics stored in a **chp_results** record on *stdout*.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

USE:
```
void chp_print_stats(record)
   chp_results *record;
```

NOTES: The output is identical to that produced by **chp_stats()**.

record: points to the **chp_results** record to print.

FUNCTION: **chp_print_trace()**

PURPOSE: Prints the results of a heap trace.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

USE:

```
void chp_print_trace(header,mode)
   chp_trace_hdr *header;
   chp_print_mode mode;
```

NOTES: *header:* points to a **chp_trace_hdr** structure at the beginning of a transcript buffer. This pointer is returned by the call to **chp_trace()**.

mode: determines which transactions will be printed. *mode* can have any of the values of the **chp_print_mode** enumeration:

```
typedef enum{
   chp_print_full,
   chp_print_unbalanced,
   chp_print_edited
}chp_print_mode;
```

If *mode* is *chp_print_full*, every heap transaction is reported. If *mode* is *chp_print_unbalanced*, only those heap transactions which were unbalanced are printed; that is, only those in which a block was allocated but not freed, or freed without having been allocated during the time the trace was active. If *mode* is *chp_print_edited*, all the unbalanced transactions except those requested by the trace code itself are printed. Allocations for transcript buffers are by definition unbalanced, since they are never freed until **chp_free_trace()** is called, at which point the transcript can no longer be printed.

The modes *chp_print_unbalanced* and *chp_print_edited* are very similar. An edited transcript is more useful if you are concerned only with which of your functions are performing unbalanced heap operations; an unbalanced transcript accounts for every unbalanced heap transaction.

Although only one heap trace may be active at a time, **chp_print_trace()** can be called any number of times to display the transcript of the ongoing trace's current state.

chp_print_trace() throws *CGEN_ERR_BADARG* if the value of *mode* is not of type **chp_print_mode**. **chp_print_trace()** cannot test its header pointer, so you must ensure its validity.

STRUCTURE: **chp_results**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

DEFINITION:

```
typedef struct{
   int
      alloc_bytes, alloc_buffers, alloc_largest,
      free_bytes, free_blocks, free_largest;
}chp_results;
```

NOTES: **chp_results** is filled in by the function **chp_statistics()** and contains heap usage statistics.

alloc_bytes: the total number of bytes allocated but not freed in the system. Each call to **cvc_alloc()** or **malloc()** increases this number by the number of bytes requested plus at least six bytes for overhead. A call to **realloc()** increases or decreases this number depending on the size of the argument.

Each time a function, variable, or macro is defined, this number increases. You can examine this number to see if the code you have written is using memory and not returning it to the heap by calling **chp_statistics()** or **chp_stats()** before and after your function. This number should remain unchanged if you are not using memory permanently.

Note: The V compiler does many hidden calls to **alloc()** and **free()**. Therefore, it is better to test compiled code since compiled code contains no hidden heap transactions.

alloc_buffers: the number of buffers allocated in the system. Each call to **cvc_alloc()** or **malloc()** increases this number by one. Each call to **free()** decreases it by one.

alloc_largest: the largest argument to **cvc_alloc()**, **malloc()**, or **realloc()**, plus at least six bytes of buffer overhead.

free_bytes: the number of bytes unallocated and available to **cvc_alloc()**, **malloc()**, and **realloc()**. Calls to the allocation routines decrease this number, calls to **free()** increase this number.

free_blocks: reflects the number of sections into which the main heap has been fragmented. This number may or may not increase as the heap is used. Free areas of memory may be tucked in between two allocated blocks; thus the heap can only satisfy allocation calls less than or equal to the size of the largest available block, as described below.

The speed of the allocation routines decreases as this number increases, because the routines search through the list of free blocks for a fragment large enough to satisfy an allocation call.

Each call to **free()** may increase this number by one, decrease it by one, or leave it unchanged, depending on how many of the two adjacent blocks can be merged with the new free block.

free_largest: the largest free fragment, or the largest buffer that can be successfully returned by the allocation routines, minus six bytes of overhead.

VARIABLE: **chp_script**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

DEFINITION:

```
chp_trace_hdr *chp_script;
```

NOTES: The global variable *chp_script* indicates whether or not a trace is active. If a trace is active, *chp_script* is set to the address of a **chp_trace_hdr** allocated by **chp_trace().**

The functions **malloc()** and **free()** know that a heap trace is active by testing *chp_script*. If *chp_script* is non-NULL, a trace is active and *chp_script* contains the address of the currently running trace's header.

When **chp_trace()** is called to stop a trace, it clears *chp_script* and returns the **chp_trace_hdr** address previously recorded there. Without *chp_script*, the address of the trace header is lost unless it was previously recorded; therefore you should save the address of the trace header when the trace is begun, so that in the event that the trace is aborted due to an error condition, the transcript can be printed and the records freed.

FUNCTION: **chp_statistics()**

PURPOSE: Scans the heap and fills in a **chp_results** structure with heap usage statistics.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

USE:

```
chp_results *chp_statistics(record)
   chp_results *record;
```

NOTES: The data in the results record is available for use by programs, or may be retained for printing at any time. Following a call to **chp_statistics()** with a call to **chp_print_stats()** will print the usage statistics on *stdout*.

record: points to the **chp_results** structure to be filled in. If the pointer is NULL, **chp_statistics()** allocates a record. In this case, however, the allocation affects the heap test results. Therefore, it is generally better to allocate all **chp_results** structures *before* running any heap tests, so that the tests reflect only changes in the code being tested.

FUNCTION: **chp_stats()**

PURPOSE: Scans the heap and prints usage statistics.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp.h*

USE:

```
void chp_stats();
```

NOTES: Calling **chp_stats()** is equivalent to calling **chp_statistics()** followed by **chp_print_stats()**. When **chp_stats()** is called, however, the statistics are printed immediately on *stdout*. Also, the results structure is allocated on the stack, and is deallocated upon return; therefore **chp_stats()** does not affect the heap statistics.

FUNCTION: **chp_trace()**

PURPOSE: Starts or stops heap tracing.

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

USE:

```
chp_trace_hdr *chp_trace(header,depth,mode)
    chp_trace_hdr *header;
    int depth, mode;
```

NOTES:

header: points to a **chp_trace_hdr**. If *header* is NULL when starting a trace, a new transcript buffer is allocated and the trace begins there. If *header* is supplied when starting a trace, **chp_trace()** will restart the trace having the header that was passed rather than begin a new trace. If *header* contains the address of a **chp_trace_hdr** and no other trace is active, tracing resumes on that transcript where it last left off. Since the header contains a pointer to the last recorded entry, no other action need be taken to restart it.

If a trace is restarted, the options in effect when it was first begun will be used. Different specifications (a different *depth* value or a function arguments request) will be ignored.

If the trace specified by *header* is currently running, no action is taken. If a different trace is currently running, the current trace is aborted and *CHP_ERR_TRACEON* is thrown.

If the trace is being stopped, *header* is ignored.

depth: specifies the depth of the stack trace. If *depth* is zero, no stack trace is performed. If *depth* is a positive integer, the stack trace will not exceed the specified number of functions. If *depth* is -1, a full stack trace is performed. If *depth* is less than -1, *CGEN_ERR_BADARG* is thrown.

chp_trace() records the same exception as **cd_strace()**, such as "bad *a6 pointer"* or "*stack bottom"*. An additional exception is "*chp_trace_alloc"*, which means that a heap allocation was needed by the trace code itself for another 8K buffer in which to record heap trace data.

If the stack trace encounters a Cognex internal function, it will be reported as "(internal function)", followed by the function's arguments if the arguments option was requested.

mode: specifies whether the trace is to be turned on or off and whether function arguments should be recorded if a new stack trace is requested. The desired mode is set by ORing various bit flags together. The flags are:

- *chp_trace_start:* Begin tracing. Only one trace may be active at any time. If no trace is active, one is started. If the current trace is already active, the trace continues and no errors are thrown. If another trace is active, it is halted and the error *CHP_ERR_TRACEON* is thrown.

- *chp_trace_stop:* Stop a trace. The currently-active trace is turned off by clearing *chp_script* and returning the transcript pointer recorded there. If *chp_script* was NULL (no trace active), NULL is returned but no error is thrown. The *header* and *depth* arguments are ignored.

  Halted traces can be restarted (and halted again) arbitrarily. It is not necessary to stop a trace in order to print its results. You can call **chp_print_trace()** to print the contents of a heap trace transcript without disabling the trace and without adverse side effects (i.e., it performs no heap transactions which would be reflected in the trace).

  When **chp_trace()** is called to stop a trace, it clears *chp_script* and returns the **chp_trace_hdr** address previously recorded there. This value is passed to **chp_print_trace()** to print trace results, and to **chp_free_trace()** to free the transcript buffer(s).

  Note: You should save the address of the trace header when the trace is begun, so that in the event the trace is aborted, the transcript can be printed and the records freed. When a trace is aborted, *chp_script* is cleared. Without *chp_script*, the address of the trace header is lost unless it was previously recorded. You will also have one or more 8K transcript buffers which can never be freed. You must free the transcript buffer by calling **chp_free_trace()**.

- *chp_trace_args:* Record function arguments. During stack tracing, function arguments as well as addresses will be recorded.

  The *chp_trace_args* flag has no meaning except when ORed with *chp_trace_start*. If used by itself or with *chp_trace_stop* it will be ignored.

STRUCTURE: **chp_trace_entry**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

DEFINITION:

```
typedef struct chp_trace_entry{
   struct chp_trace_entry *next;
   int block_size, block_addr,
       merge_high, merge_low;
   chp_entry_status status;
}chp_trace_entry;
```

NOTES: **chp_trace_entry** is the data type describing a single entry in a heap trace transcript buffer. This description is provided for informative purposes only. You should never need to modify a **chp_trace_entry**.

next: points to the next **chp_trace_entry** (whether part of this buffer or not).

block_size: the size of the heap block (not including the block header) described by the entry.

block_addr: the address (not including the header) of the heap block described by the entry.

merge_high and *merge_low:* the addresses of the adjoining blocks on the free list. These addresses are recorded only if the block described by the entry is merged with them when it is freed.

status: a structure that records:

- whether or not a particular transaction was a **free()** (1) or a **malloc()** (0);
- whether it was a balanced (1) or unbalanced (0) transaction; and
- whether it was requested by the heap trace code itself for a new transcript buffer (1) or not (0).

```
typedef struct{
   unsigned int free: 1;
   unsigned int balanced: 1;
   unsigned int new_buffer: 1;
}chp_entry_status;
```

Once a trace is started, changing its header guarantees that the trace results will be corrupted. If you want new options, you must halt the old trace and start a new one.

STRUCTURE: **chp_trace_hdr**

REFERENCE: System Software: Memory

INCLUDE FILE: *chp_trac.h*

DEFINITION:

```
typedef struct{
   struct chp_trace_entry *current;
   int depth,bottom,high_addr,low_addr;
   chp_hdr_status status;
}chp_trace_hdr;
```

NOTES: The heap tracing functions use several variables to specify their mode of operation and to manage the transcript of the trace. For convenience, these variables are kept in a header structure of type **chp_trace_hdr**. This description is provided for informative purposes only. You should never need to modify a **chp_trace_hdr**.

current: points to the last *chp_trace_entry* recorded.

depth: the depth of the stack trace. It is set to zero if no stack trace is to be performed. It is set to -1 if a full stack trace is to be performed.

bottom: the first address beyond the end of the current transcript buffer, and is recorded to keep the trace from overwriting unallocated memory.

high_addr and *low_addr:* reserved for future use.

status: records whether or not arguments will be recorded during stack tracing. If the value of *status* is nonzero, arguments are recorded during the stack trace. *status* is implemented as a structure containing a bit field.

```
typedef struct{
   unsigned int args_opt: 1;
}chp_hdr_status;
```

Once a trace is started, changing its header guarantees that the trace results will be corrupted. If you want new options, you must halt the old trace and start a new one.

If a trace is active, the global pointer *chp_script* is set to the address of a **chp_trace_hdr** allocated by **chp_trace()**:

```
extern chp_trace_hdr *chp_script;
```

If no trace is active, *chp_script* is NULL.

UNION: **cia_15_8**

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

DEFINITION:

```
typedef union {
   CIA_15_8 n;
   struct {
      char i;
      unsigned char f;
   } p;
} cia_15_8;
```

NOTES: The vision processor stores some fixed point numbers in the union **cia_15_8**. This union has lower precision than **cia_31_16** and **cia_15_14**, but has a similar form.

n: a **CIA_15_8** (16 bit signed integer) which represents the entire fixed point number.

p: a **structure** containing *i*, which is a signed **char** representing the integer portion of the fixed point number; and *f*, which is an unsigned **char** representing the 8 bit fractional portion of the fixed point number.

UNION: **cia_31_16**

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

DEFINITION:

```
typedef union {
   CIA_31_16 n;
   struct {
      short i;
      unsigned short f;
   } p;
} cia_31_16;
```

NOTES: The vision processor stores some fixed point numbers as the union **cia_31_16.**

n: a **CIA_31_16** (a 32 bit integer, including the sign bit) which represents the entire fixed point number.

p: a **structure** containing *i,* which is a signed **short** representing the integer portion of the fixed point number; and *f,* which is an unsigned **short** representing the 16 bit fractional portion of the fixed point number.

FUNCTION: **cia_atan()**

PURPOSE: Returns the arctangent of the specified fraction as an integer number of tenths of a degree.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:

```
int cia_atan (numer, denom)
   int numer, denom;
```

NOTES: *numer:* the numerator of the fraction.

denom: the denominator of the fraction.

cia_atan() throws the error *CGEN_ERR_BADARG* if the value of either *numer* or *denom* is outside of the range 32767 to -32768.

STRUCTURE: **cia_coeff_matrix**

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_coef.h*

DEFINITION:

```
typedef struct cia_coeff_matrix {
   cia_31_16
      xx, yx,
      xy, yy,
      x0, y0;
} cia_coeff_matrix;
```

NOTES: **cia_coeff_matrix** contains the coefficients used in the transformation equations. It is defined internally to *V* and is used in the structure **cpsi_geom**.

xx, yx, xy, yy, x0, and y0: the coefficients of the transformation equations. *x0* and *y0* represent the translation of the source to the target coordinate system.

FUNCTION: **cia_coord_xform()**

PURPOSE: Converts a specified point from one coordinate system to another coordinate system using the transformation equations described below.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_coef.h*

USE:

```
void cia_coord_xform(x,y,x_p,y_p,cf_p)
   int x, y;
   int *x_p, *y_p;
   cia_coeff_matrix *cf_p;
```

NOTES: *x* and *y:* the x and y coordinates of the point to be transformed.

x_p and *y_p:* point to the integer values of the transformed point. **cia_coord_xform()** calculates *x_p* and *y_p*.

The transformation equations used are:

$$xp = (x \cdot xx\ \ coefficient) + (y \cdot yx\ \ coefficient) + (x0\ \ coefficient)$$

$$yp = (x \cdot xy\ \ coefficient) + (y \cdot yy\ \ coefficient) + (y0\ \ coefficient)$$

where xx, yx, xy, yy, x0 and y0 describe a transformation.

cf_p: points to the previously allocated **cia_coeff_matrix** structure that contains the coefficients to the transformation equations.

FUNCTION: **cia_cos()**

PURPOSE: Returns the cosine of the specified angle as a 16-bit signed integer (**CIA_15_14**).

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
CIA_15_14 cia_cos(angle)
   int angle;
```

NOTES: *angle:* the angle for which the cosine is returned. *angle* should be expressed as an integer number of tenths of a degree. *angle* need not be in the range 0 to 3600, because **cia_cos()** reduces angles outside of this range to the 0 to 3600 range. Therefore, if you do a calculation with incremental values of *angle* you need not worry about *angle* increasing beyond the range of the function.

FUNCTION: **cia_cos16()**

PURPOSE: Computes the cosine of a **CIA_31_16** number and returns a signed 16 bit integer (**CIA_15_14**).

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
CIA_15_14 cia_cos16(angle)
   CIA_31_16 angle;
```

NOTES: *angle:* the angle for which the cosine is returned.

The result is accurate to 13 bits for any angle supplied.

MACRO: **CIA_DOUBLE_TO_8()**

PURPOSE: Converts a floating point number to a **CIA_15_8.**

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
#define CIA_DOUBLE_TO_8(x) ((short) ((x)*256.))
```

NOTES: This macro returns a **CIA_15_8** number.

x: the double to convert.

MACRO: **CIA_DOUBLE_TO_16()**

PURPOSE: Converts a floating point number to a **CIA_31_16**.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:

```
#define CIA_DOUBLE_TO_16(x) ((int) ((x)* 65536.))
```

NOTES: This macro returns a **CIA_31_16**.

x: the double to convert.

STRUCTURE: **cia_float**

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

DEFINITION:

```
typedef struct {
   short e;
   unsigned short m;
} cia_float;
```

NOTES: **cia_float** is an obsolete fast floating point number format, consisting of an exponent and a mantissa. It is used only in the *Searching* option to represent certain search results.

e: the exponent portion of the number.

m: the mantissa portion of the number.

You can convert a **cia_float** to a double using the macro **ff_double()**:

```
#define ff_double(n) (ldexp(double) (n).m, (n).e))
```

MACRO: **CIA_FLOAT8()**

PURPOSE: Converts a **CIA_15_8** to a double precision floating point number.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
#define CIA_FLOAT8(x) ((x)/256.)
```

NOTES: This macro returns a **double**.

x: the **CIA_15_8** number to convert.

MACRO: **CIA_FLOAT14()**

PURPOSE: Converts a **CIA_15_14** to a double precision floating point number.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
#define CIA_FLOAT14(x) ((x)/16384.)
```

NOTES: This macro returns a **double**.

x: the **CIA_15_14** number to convert. Numbers of this form are returned by the functions **cia_sin()**, **cia_sin16()**, **cia_cos()**, and **cia_cos16()**.

MACRO: **CIA_FLOAT16()**

PURPOSE: Converts a **CIA_31_16** to a double precision floating point number.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
#define CIA_FLOAT16(x) ((x)/65536.)
```

NOTES: This macro returns a double.

x: the **CIA_31_16** number to convert.

FUNCTION: **cia_limit()**

PURPOSE: Takes the arguments *x, x_low,* and *x_high,* and returns *x* if *x* lies between *x_low* and *x_high.* Otherwise it returns the limit nearest to *x.* All arguments to **cia_limit()** must be integers.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
int cia_limit(x,x_low,x_high)
   int x, x_low, x_high;
```

NOTES: *x:* the value to be compared to the limits.

x_low: the lower limit.

x_high: the upper limit.

cia_limit() throws the error *CGEN_ERR_BADARG* if *x_low* is greater than *x_high.*

FUNCTION: **cia_lin_limit()**

PURPOSE: Limits the specified variable to a certain range in the x and y dimensions.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
int cia_lin_limit(x,x_low,x_high,y_low,y_high)
   int x, x_low, x_high, y_low, y_high;
```

NOTES: The variable can have values that vary linearly between the limits.

x: the value to compare to the limits.

x_low: the lower limit values can have in the x dimension.

x_high: the upper limit values can have in the x dimension.

y_low: the lower limit values can have in the y dimension.

y_high: the upper limit values can have in the y dimension.

cia_lin_limit() transforms *x* into *f(x)* such that

if $x < x_low$, $f(x) = f(x_low) = y_low$

if $x > x_high$, $f(x) = f(x_high) = y_high$

if $x_low < x < x_high$, $f(x)$ lies linearly between y_low and y_high

cia_lin_limit() returns *y_low* if *x* is less than or equal to *x_low*. It returns *y_high* if *x* is greater than or equal to *x_high*. If *x* is between *x_low* and *x_high*, the value returned is:

(x - x_low * ((y_high - y_low)/(x_high-x_low))) + y_low

STRUCTURE: **cia_point**

REFERENCE: Image Processing: Images

INCLUDE FILE: *cia_def.h*

DEFINITION:

```
typedef struct {
   int x, y;
} cia_point;
```

NOTES: **cia_point** defines the (x,y) coordinates of a point in an image.

x, y: the x and y coordinates of the point.

STRUCTURE: **cia_rect**

REFERENCE: Image Processing: Images

INCLUDE FILE: *cia_def.h*

DEFINITION:

```
typedef struct {
   cia_point ul;
   short w, h;
} cia_rect;
```

NOTES: **cia_rect** defines a rectangular area, typically used to define a window of an image buffer.

ul specifies the upper left corner of the rectangle.

w, h: the width and height of the rectangle.

MACRO: **CIA_RND8()**

PURPOSE: Rounds a **CIA_15_8** to the nearest integer.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
#define CIA_RND8(x) (((x)+0x80) >> 8)
```

NOTES: This macro returns an integer.

x: the **CIA_15_8** number to round.

MACRO: **CIA_RND16()**

PURPOSE: Rounds a **CIA_31_16** to the nearest integer.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
#define CIA_RND16(x) (((x) + 0x8000) >> 16)
```

NOTES: This macro returns an integer.

x: the **CIA_31_16** number to convert.

FUNCTION: **cia_sin()**

PURPOSE: Returns the sine of the specified angle as a 16-bit signed integer (**CIA_15_14**).

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
CIA_15_14 cia_sin(angle)
   int angle;
```

NOTES: *angle:* the angle for which the sine is returned. *angle* should be an integer number in tenths of a degree. *angle* need not be in the range 0 to 3600, because **cia_sin()** reduces angles outside of this range to the 0 to 3600 range. Therefore, if you do a calculation with incremental values of *angle*, you need not worry about *angle* increasing beyond the range of the function.

FUNCTION: **cia_sin16()**

PURPOSE: Computes the sine of a **CIA_31_16** number and returns a signed 16 bit (**CIA_15_14**).

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
CIA_15_14 cia_sin16(angle)
   CIA_31_16 angle;
```

NOTES: *angle:* the angle for which the sine is returned.

The returned value is accurate to 13 bits for any angle supplied. The execution time for this function is approximately the same as that for **cia_sin()**.

FUNCTION: **cia_sqrt()**

PURPOSE: Returns the unsigned integer square root of its argument.

REFERENCE: System Software: Mathematics

INCLUDE FILE: *cia_def.h*

USE:
```
int cia_sqrt(n)
   unsigned int n;
```

NOTES: Any fractional value is truncated, not rounded.

n: the number for which the square root is returned.

FUNCTION: **cif_create_pd()**

PURPOSE: Allocates a **cif_pd** data structure on the heap, clears it, and returns a pointer to the allocated structure.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
cif_pd *cif_create_pd();
```

NOTES: **cif_create_pd()** does not allocate storage for the *function_iv* or *device_iv* instance variables, since their sizes vary. It initializes *cif_pd.xform.xx* and *cif_pd.xform.yy* to 1, specifying the identity transform.

If you are creating a pointing device for one of the four standard purposes, you can call a Cognex supplied function that allocates a **cif_pd** structure and fills it in with default values for that device application. For example, to create a pointing device for drawing windows, call **cif_make_window()** which allocates a **cif_pd** structure and fills it in with the values needed for a window drawing pointing device.

If you are using your own pointing device, you must fill in the **cif_pd** with the addresses of the *track, start, check, update*, and *stop* routines to use. These may be routines you have written or Cognex routines. You also must define **typedefs** and allocate storage for any necessary *function_iv* and *device_iv.*

FUNCTION: **cif_cursor_stop()**

PURPOSE: Reports the final cursor coordinates before exiting the cursor *track* routine.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_cursor_stop(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* the final cursor horizontal coordinate.

y: the final cursor vertical coordinate.

p_pd: points to the cursor **cif_pd** whose position is reported.

cif_cursor_stop() prints the following upon completion:

cursor at (<x>,<y>)

FUNCTION: **cif_cursor_update()**

PURPOSE: Updates the cursor on the screen.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_cursor_update(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: **cif_cursor_update()** clears the old cursor from the screen, then draws the new cursor.

x: the new horizontal cursor coordinate.

y: the new vertical cursor coordinate.

p_pd: points to a **cif_pd** structure. The cursor is drawn in the image buffer specified by *p_pd->display*.

cif_cursor_update() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_delete_pd()**

PURPOSE: Deletes a **cif_pd** structure and frees the memory allocated for the structure.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:
```
cif_delete_pd(p_pd)
   cif_pd *p_pd;
```

NOTES: **cif_delete_pd()** also frees the memory allocated for the *function_iv* and *device_iv* in the **cif_pd**.

- *p_pd:* points to the **cif_pd** to delete.

Freed memory is returned to the heap. **cif_delete_pd()** does not delete the pointing device functions or structures pointed to by *p_pd->caq_params* and *p_pd->display*.

VARIABLE: **cif_fg_cursor**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
cif_pd *cif_fg_cursor;
```

NOTES: Points to the **cif_pd** that controls the standard Cognex cursor. It is used to move a cursor on the frame grabber image. *cif_fg_cursor* is created when you **#include** *<cif.v>*. *cif_fg_cursor* is designed for use with the Cognex 3-button trackball. Moving the trackball moves the cursor on the screen; hitting any button exits the trackball routine.

When *cif_fg_cursor* is created:

- *cif_fg_cursor->caq_params* is set to the global *caq_context* pointer *caq_mode*. Another *caq_context* can be substituted for the *caq_mode* pointer by modifying *cif_fg_cursor->caq_params*. However, the *track*, *start*, *check*, and *update* routines for *cif_fg_cursor* all require that *cif_fg_cursor->display* points to the frame grabber.
- *cif_fg_cursor->display* is set to *caq_image*, the frame grabber.

VARIABLE: **cif_fg_gainoff**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
cif_pd *cif_fg_gainoff;
```

NOTES: *cif_fg_gainoff* points to the standard Cognex gain and offset pointing device. It is used to select new values for gain and offset. This pointing device is designed for use with the Cognex 3-button trackball.

Moving the trackball vertically changes the camera gain, moving it horizontally changes the camera offset. The new gain and offset values are recorded in the **caq_context** pointed to by *cif_fg_gainoff->caq_params->gain* and *cif_fg_gainoff->caq_params->offset*. Hitting any button exits the trackball routine.

When *cif_fg_gainoff* is created:

- *cif_fg_gainoff->caq_params* is set to the global **caq_context** pointer *caq_mode*.

  Other **caq_context** structures can be substituted for *caq_mode* by modifying *cif_fg_gainoff->caq_params*. The *start* routine for *cif_fg_gainoff* assumes that *cif_fg_gainoff->display* points to the frame grabber.

- *cif_fg_gainoff->display* is set to *caq_image*, the frame grabber pointer.

VARIABLE: **cif_fg_split**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
cif_pd *cif_fg_split;
```

NOTES: *cif_fg_split* points to the standard Cognex screen-splitting **cif_pd**. It is used to select the line at which the screen is split to display input from two cameras. This pointing device is designed for use with the Cognex 3-button trackball. Moving the trackball vertically splits the image displayed on the monitor between two cameras. The trackball uses the *camera* and modifies the *split* specified in the **caq_context** pointed to by *cif_fg_split->caq_params*.

The value of *split* can range from 0 to *CAQ_HEIGHT*, the number of lines in the frame grabber.

If *split* is zero, the displayed image consists entirely of the view from the camera specified by *cif_fg_split->caq_params->camera*. *camera* must be a camera number from 1 to 4.

If *split* is nonzero, it represents the line at which the image is split between two cameras. In this case, the *cif_fg_split->caq_params->camera* value must be either one or three. If it is one, the split is between cameras one and two; if it is three, the split is between cameras three and four.

When *cif_fg_split* is created:

- *cif_fg_split->caq_params* is set to the global **caq_context** pointer *caq_mode*.

  Other **caq_context** structures can be substituted for *caq_mode* by modifying *cif_fg_split->caq_params*. The *track*, *start*, *check*, and *update* routines all require that *cif_fg_split->display* points to the frame grabber.

- *cif_fg_split->display* is set to *caq_image*, the frame grabber pointer.

VARIABLE: **cif_fg_window**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
cif_pd *cif_fg_window;
```

NOTES: *cif_fg_window* is a pointer to the standard Cognex window **cif_pd**. It is used to change the size or position of a window of the frame grabber. This **cif_pd** is designed for use with the Cognex 3-button trackball.

cif_fg_window has two modes: position and size. The values of the *x* and *y* variables depend on the mode. In position mode, *x* and *y* are the current location of the window origin. In size mode, *x* and *y* are the width and height of the window, respectively.

When *cif_fg_window* is created:

- *cif_fg_window->function_iv* points to a **cif_iv_window** structure. This structure consists of a window *color* component (black, white, blinking, or invisible), the trackball *mode* (position or size, described above), and a pointer, *imbuf*, to the frame grabber window in which the window is being drawn.
- *cif_fg_window->x* is the current position of the window origin if the trackball is in position mode. It is the width of the window if the trackball is in size mode.
- *cif_fg_window->y* is the current y position of the window origin if the trackball is in position mode. It is the window height if the trackball is in size mode.

  cif_window_update() treats *x* and *y* as having the values of the last selected mode.

  Each time you call **cif_window_stop()**, *cif_fg_window->x* and *cif_fg_window->y* are set to the current window position. However, if you substitute a different stop routine, the values assigned to *x* and *y* are specified by the update routine and, unless you assign them, may be either the coordinate pair *or* the width and height of the window.
- *cif_fg_window->caq_params* is set to the global **caq_context** pointer *caq_mode*.

  Other **caq_context** structures can be substituted for *caq_mode* by modifying *cif_fg_split->caq_params*. The *track, start, check*, and *update* routines all require that *cif_fg_window->display* points to the frame grabber.
- *cif_fg_window->display* is set to *caq_image*, the frame grabber pointer.

The pointer to the trackball window is frequently used as an argument to vision functions such as searching or training.

FUNCTION: **cif_gainoff_start()**

PURPOSE: Initialization routine for a gain and offset trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_gainoff_start(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x* and *y:* currently unused; they are passed so that all *start* functions maintain the same calling convention.

p_pd: points to the **cif_pd** structure to initialize.

cif_gainoff_start() copies *p_pd->caq_params->gain* to *p_pd->y* and copies *p_pd->caq_params->offset* to *p_pd->x*, then calls **cif_realtime()** to put the frame grabber in real-time mode.

cif_gainoff_start() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_gainoff_stop()**

PURPOSE: Reports the final gain and offset before exiting the gain and offset *track* routine.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_gainoff_stop(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* the current offset.

y: the current gain.

p_pd: points to the **cif_pd** structure from which to get the current gain and offset.

cif_gainoff_stop() prints the following upon completion:

```
offset = <x>, gain = <y>
```

where *x* is the current offset and *y* is the current gain.

FUNCTION: **cif_gainoff_update()**

PURPOSE: Updates the gain and offset.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_gainoff_update(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* the new offset. **cif_gainoff_update()** sets both *p_pd->caq_params->offset* and *p_pd->x* to *x*.

y: the new gain. **cif_gainoff_update()** sets both *p_pd ->caq_params->gain* and *p_pd->y* to *y*.

p_pd: points to the **cif_pd** structure being updated.

cif_gainoff_update() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

STRUCTURE: **cif_iv_cursor**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
typedef struct{
   int
      x_arm, y_arm,
      color;
}cif_iv_cursor;
```

NOTES: **cif_iv_cursor** specifies the function instance variables for a pointing device that controls a cursor.

x_arm: the length of the cursor's horizontal arm.

y_arm: the length of the cursor's vertical arm.

color: the color of the cursor. Valid colors are: *CAQ_WHITE* (white), *CAQ_BLACK* (black), *CAQ_BLINK* (alternating black and white), and *CAQ_PASS* (the image shows through).

To refer to the contents of a **cif_iv_cursor** structure, it is necessary to do a cast, because *p_pd->function_iv* is defined as a pointer to a **char**, so it may point to anything. For example, the cursor color can be changed to white as follows:

```
((cif_iv_cursor *) p_pd->function_iv)->color = CAQ_WHITE;
```

where *p_pd* is a pointer to a cursor pointing device.

STRUCTURE: **cif_iv_tb3**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
typedef struct{
   short buttons;
}cif_iv_tb3;
```

NOTES: **cif_iv_tb3** is the instance variable for the Cognex 3-button trackball. It contains an element describing the state of the three buttons. *Track* routines monitor the state of the trackball buttons and use this state for control information. For the functions that use the Cognex 3-button trackball, only the low-order three bits are of interest.

buttons: the state of the buttons on the trackball device.

The three lowest bits reflect the state of the trackball buttons. Bit zero corresponds to the left button, bit one to the middle button, and bit two to the right button. If a button is up (not depressed), the corresponding bit is cleared. If a button is down (depressed), the corresponding bit is set.

To refer to the contents of a **cif_iv_tb3** structure, it is necessary to do a cast, because the device-specific instance variable pointer *p_pd->device_iv* is defined as a pointer to a **char** and could point to anything. For example, to see if any of the buttons have been pressed:

```
if (((cif_iv_tb3 *)p_pd->device_iv)->buttons)
```

where *p_pd* points to a pointing device that uses the Cognex 3-button trackball.

STRUCTURE: **cif_iv_window**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
typedef struct{
   int color,mode;
   cip_buffer *imbuf;
}cif_iv_window;
```

NOTES: **cif_iv_window** is the instance variable for a window pointing device.

color: the color in which the window border is outlined on the monitor. Valid colors are: *CAQ_WHITE* (white), *CAQ_BLACK* (black), *CAQ_BLINK* (alternating black and white), and *CAQ_PASS* (the image shows through).

mode: the window trackball mode. The trackball can be in either *position* or *size* mode. In position mode, moving the trackball moves the window on the screen. In size mode, moving the trackball expands or contracts the right side of the window, the bottom, or both, depending on the direction in which the trackball is rolled.

Specifying zero for *mode* puts the trackball into size mode, specifying one puts it into position mode. The trackball defaults to position mode when a window is started.

imbuf: points to the image buffer in which the window will be drawn. The Cognex default window trackball **cif_fg_window** assumes this is a window into the frame grabber.

When a window trackball is started, it begins in *position mode* (*cif_iv_window.mode* is set to one). Moving the trackball moves the window on the screen. In *size mode*, the window size changes; the upper left corner of the window remains stationary while the right and bottom sides shrink or grow, depending on the direction of trackball movement. *cif_iv_window.mode* is zero in *size mode*.

The middle trackball button switches between *position* and *size modes*. Hitting the left button exits the window track routine. Hitting the right button has no effect.

To refer to the contents of a **cif_iv_window** structure, it is necessary to do a cast, because *p_pd->function_iv* is defined as a pointer to a **char**, so it may point to anything. For example, the window's color can be changed to white as follows:

```
((cif_iv_window *) p_pd->function_iv)->color = CAQ_WHITE;
```

where *p_pd* is a pointer to a window pointing device.

FUNCTION: **cif_magnify()**

PURPOSE: Magnify a window into the frame grabber by the specified factor.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_magnify(mag,p_pd)
   int mag;
   cif_pd *p_pd;
```

NOTES: *mag:* the factor by which to magnify the window.

p_pd: points to the **cif_pd** whose *function_iv->imbuf* window is to be magnified.

cif_magnify() calls the functions **caq_acquire()** and **cgr_display_image()** and may throw any errors thrown by those functions.

FUNCTION: **cif_make_cursor()**

PURPOSE: Allocates and initializes a **cif_pd** structure for a standard cursor.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
cif_pd *cif_make_cursor(p_image,p_caq_context)
   cip_buffer *p_image;
   caq_context *p_caq_context;
```

NOTES: **cif_make_cursor()** also allocates a **cif_iv_cursor** structure and a **cif_iv_tb3** (the Cognex 3-button trackball) structure.

p_image: points to the image buffer in which to display the cursor.

p_image is provided for users who want to use a trackball in an image buffer other than the frame grabber. However, the cursor *start* and *update* routines are written for use with the frame grabber (*caq_image*), so to use an arbitrary **cip_buffer** you must create your own versions of these functions.

p_caq_context: points to the **caq_context** containing the acquire parameters for the cursor.

The allocated **cif_iv_cursor** structure is initialized as follows:

- *color* is set to *CAQ_BLACK* (black).

The components of the returned **cif_pd** are initialized as follows:

- *caq_params* is set to *p_caq_context*, if a *p_caq_context* is passed, or to the global default *caq_mode* if *p_caq_context* is NULL.
- *display* is set to *p_image*, if a *p_image* is passed, or to the frame grabber pointer *caq_image* if *p_image* is NULL. The cursor is positioned at the center of the *display* image buffer.
- *track* points to the function **cif_pd_track()**.
- *start* points to the function **cif_pd_start()**.
- *check* points to the function **cif_pd_check()**.
- *update* points to the function **cif_cursor_update()**.
- *stop* points to the function **cif_cursor_stop()**.

FUNCTION: **cif_make_gainoff()**

PURPOSE: Allocates and initializes a **cif_pd** structure to manipulate camera gain and offset and returns a pointer to the structure.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
cif_pd *cif_make_gainoff(p_image,p_caq_context)
   cip_buffer *p_image;
   caq_context *p_caq_context;
```

NOTES: **cif_make_gainoff()** also allocates a **cif_iv_tb3** structure.

p_image: points to the image buffer to display while the gain and offset are modified.

p_caq_context: points to the **caq_context** to modify.

cif_make_gainoff() initializes the components of the returned **cif_pd** structure as follows:

- *device_iv* is set to point to the allocated **cif_iv_tb3** structure.
- *function_iv* is set to NULL.
- *caq_params* is set to *p_caq_context*. If no *p_caq_context* is passed, *caq_params* is set to the global default *caq_mode*.
- *display* is set to *p_image*. If *p_image* is NULL, *display* is set to the frame grabber pointer *caq_image*. *p_image* is provided for users who want to use the trackball in an image other than the frame grabber. However, **cif_gainoff_start()** and **cif_gainoff_update()** are written for use with the frame grabber (*caq_image*), so to use a different **cip_buffer** you must create your own *start* and *update* routines.
- *track* points to the function **cif_pd_track()**.
- *start* points to the function **cif_gainoff_start()**.
- *check* points to the function **cif_pd_check()**.
- *update* points to the function **cif_gainoff_update()**.
- *stop* points to the function **cif_gainoff_stop()**.

FUNCTION: **cif_make_split()**

PURPOSE: Creates and initializes a **cif_pd** structure for splitting the displayed image between two cameras.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:
```
cif_pd *cif_make_split(p_image,p_caq_context)
   cip_buffer *p_image;
   caq_context *p_caq_context;
```

NOTES: **cif_make_split()** returns a pointer to the **cif_pd** structure. **cif_make_split()** also allocates a **cif_iv_tb3** structure.

p_image: points to the **cip_buffer** in which to display the split screen.

p_image is for users who want to use an image buffer other than the frame grabber for trackball manipulations. However, the *start* and *update* routines are written for use with the frame grabber (*caq_image*), to use an arbitrary **cip_buffer** you must create your own versions of these functions.

p_caq_context points to the **caq_context** that specifies the acquire parameters for split screen operations.

The components of the returned **cif_pd** structure are initialized as follows:

- *device_iv* is set to point to the allocated **cif_iv_tb3** structure.
- *function_iv* pointer is set to NULL.
- *caq_params* is set to *p_caq_context*, if one is passed, or the global default *caq_mode* if *p_caq_context* is NULL.
- *display* is set to *p_image*, if one is passed, or to the frame grabber pointer *caq_image* if *p_image* is NULL.
- *y*, if greater than zero, is the line at which the split occurs. Initially, the value is zero; only one of two possible camera views displays on the monitor. The lower limit for *y* is zero; the upper limit is the height of the *display* image. If the frame grabber is used, this is equal to *CAQ_HEIGHT*.
- *x* is unused; its initial value and its limits are all zero.
- *track* points to the function **cif_pd_track()**.
- *start* points to the function **cif_split_start()**.
- *check* points to the function **cif_pd_check()**.
- *update* points to the function **cif_split_update()**.
- *stop* points to the function **cif_split_stop()**.

FUNCTION: **cif_make_window()**

PURPOSE: Creates and initializes a **cif_pd** structure for window manipulation.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
cif_pd *cif_make_window(p_image,p_caq_context,p_window)
   cip_buffer *p_image;
   caq_context *p_caq_context;
   cip_buffer *p_window;
```

NOTES: **cif_make_window()** returns a pointer to the allocated and initialized **cif_pd** structure. A **cif_iv_tb3** structure is allocated and *device_iv* is set to point to it. A **cif_iv_window** structure is allocated and *function_iv* is set to point to it.

p_image: points to the image buffer in which window operations are displayed.

p_image is provided for users who want to use an image buffer other than the frame grabber for trackball manipulations. However, the window *start* and window *update* routines are written for use with the frame grabber (*caq_image*), so to use another **cip_buffer** you must create your own version of these functions.

p_caq_context: points to the **caq_context** that specifies acquire parameters for window operations.

p_window: points to the **cip_buffer** to use; it must be a window of *p_image*. If *p_window* is NULL, a window is created and centered in the buffer specified for the *display* component of the **cif_pd**.

The returned **cif_pd** structure is initialized as follows:

- *display* is set to *p_image*, if one is passed, or to the frame grabber *(caq_image)* if *p_image* is NULL. The borders of the *display* image provide the window's initial *x* and *y* boundaries.
- *caq_params* is set to *p_caq_context*, if one is passed, or to the global default *caq_mode* if *p_caq_context* is NULL.
- *track* points to the function **cif_window_track()**.
- *start* points to the function **cif_window_start()**.
- *check* points to the function **cif_window_check()**.
- *update* points to the function **cif_window_update()**.
- *stop* points to the function **cif_window_stop()**.

Window operations only use two trackball buttons. The left button exits, and the middle button toggles between *position* and *size* modes.

FUNCTION: **cif_open_tb()**

PURPOSE: Open a serial line for use by a trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
FILE *cif_open_tb(devname)
   char *devname;
```

NOTES: *devname:* points to the full pathname of the device to open. If the pointer *devname* is NULL, */dev/ser2* is used.

When you call **cif_open_tb()**, **fopen()** is called to open *devname*, and the line is initialized to operate at 9600 baud in **raw** mode with **echo** off. **cif_open_tb()** usually need be called only once, at setup time, and is done automatically when you **#include** the file *<cif.v>*.

The value returned by **cif_open_tb()** must be used by any function that reads raw data from the trackball. The standard trackball interface functions, described below, assume that this value has been stored in the global variable *cif_tb_file*. **cif_open_tb()** does not automatically assign this value to *cif_tb_file*.

You may close the line by calling **fclose()** with the pointer returned by **cif_open_tb()**. If you close the line, you cannot resume trackball operations until another line is opened using **cif_open_tb()**.

The following code fragment illustrates a standard trackball opening routine using **cif_open_tb()**:

```
if (cif_tb_file != NULL){
   fclose(cif_tb_file);
   cif_tb_file = 0;
}
cif_tb_file = cif_open_tb(devname);
```

cif_open_tb() throws the error *CIO_ERR_FILE* if the call to **fopen()** fails.

STRUCTURE: **cif_pd**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
typedef struct{
   int x, y;
   int xlo,xhi,ylo,yhi;
   char *function_iv, device_iv;
   cia_coeff_matrix xform;
   void
      (*track)(), (*start)(), (*check)(),
      (*update)(), (*stop)();
   cip_buffer *display;
   caq_context *caq_params;
}cif_pd;
```

NOTES: The **cif_pd** data structure specifies a pointing device including:

- the variables controlled by the pointing device
- the instance variables specific to the pointing device hardware
- the instance variables specific to the pointing device application
- the methods that control the pointing device
- the image buffer in which the pointing device operates
- the frame grabber mode during which the device is used. This data structure is used for all pointing devices.

x and *y:* depending on the pointing device application, can be either:

- the image coordinates of an object controlled by the pointing device (for example, a window or a cursor)
- the values of variables controlled by the pointing device (for example the camera gain and offset).

xlo, xhi, ylo and *yhi:* depending on the pointing device application, can be:

- the limits for the location of an object controlled by the pointing device
- the limits for the values of variables controlled by the pointing device.

function_iv: points to the function instance variables (the data required for a particular pointing device application). Specify NULL if the pointing device application does not require any application specific data.

Application specific data are data required for customizing the pointing device to implement a function such as window selection or cursor positioning. For example, for a pointing device that draws the outline of a win-

dow in an image, the function instance variables must include a **cip_buffer** to keep track of the window's current size and position. In contrast, the function instance variables needed to implement a cursor are the dimensions of the cursor and the color for the cursor graphics.

If you create a pointing device to perform operations other than the four standard operations, define a C **typedef** for *function_iv*. You need this **typedef** to cast a **cif_pd** instance variable pointer before using it to reference instance variable structure components.

device_iv: points to the device instance variables (the data required for the device hardware). Specify NULL if the pointing device does not require device specific data.

If you use a pointing device other than the Cognex 3-button trackball, define a C **typedef** for *device_iv*. The **typedef** is needed so that you may cast a **cif_pd** instance variable pointer before using it to reference device instance variable structure components.

xform: the **cia_coeff_matrix** containing the data that relates the coordinate system of the pointing device to the coordinate system of the functional application. For example, with the standard cursor, since the frame grabber origin (0,0) is in the upper left corner, spinning the trackball up (increasing the trackball increments) increases the *y* coordinate, which moves the cursor down. The coordinate transformation matrix for the standard cursor has *xform.yy = -1*, passing the horizontal (*x*) trackball increments unchanged, but multiplying the vertical (*y*) increments by -1 so that the cursor moves in the same direction as the trackball.

You can use the *scale* components of a **cpsi_geom** structure to change the relationship between incremental movements of a pointing device and incremental movements of the object controlled by the pointing device, such as a cursor. By adjusting *scale*, you can make a cursor move large distances with small movements of the trackball or vice versa.

track: points to the *track* subroutine. The *track* subroutine is the primary pointing device subroutine. It calls the other subroutines (*start*, *check*, *update*, *stop*) as well as the device dependent routine that reads data from the device.

The Cognex *track* routines use the integer portion of the **cia_coeff_matrix**. You must provide your own *track* routine if your application requires a resolution that can only be achieved using the fractional part of the *xform* elements.

start: points to the *start* subroutine. *start* performs initialization, such as selecting an appropriate frame grabber mode or writing a prompt on the screen.

check: points to the *check* subroutine. *check* enforces limits on the *x* and *y* values returned by the pointing device. The check routine clamps the *x* and *y* values to limits dictated by the device application.

update: points to the *update* subroutine. *update* implements actions relevant to changes in the *x* and *y* values of the pointing device. For example, a window update routine redraws the window at its new location when the *x,y* values of the pointing device change.

stop: points to the *stop* subroutine. *stop* reports information about the final state of the pointing device or performs other post processing activities. For example, the stop routine for a pointing device that draws a window prints the current location and size of the upper left corner (origin) of the window.

display: points to the **cip_buffer** in which the pointing device is used.

caq_params: points to the **caq_context** to use for the pointing device. It may equal the global *caq_mode* if you want to use the current imaging setup or have only one imaging setup.

If you have a multi-camera setup or several sets of acquire parameters (such as *acquire_polarity,* gain, or offset) you could:

- create multiple instances of a particular pointing device each with a unique **caq_context** pointer
- change the pointer each time you use the pointing device

Update routines for **cif_pd**s that control acquisition parameters modify the corresponding values in the **caq_context** pointed to by *caq_params*.

FUNCTION: **cif_pd_check()**

PURPOSE: The generic check routine used by the Cognex 3-button trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_pd_check(p_x,p_y,p_pd)
   int *p_x, *p_y;
   cif_pd *p_pd;
```

NOTES: *p_x:* points to the x location of the trackball.

p_y: points to the y location of the trackball.

p_pd: points to the pointing device to use.

FUNCTION: **cif_pd_start()**

PURPOSE: The generic *start* routine used by the Cognex 3-button trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void start(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* the x location of the trackball.

y: the y location of the trackball.

p_pd: points to the pointing device to use.

FUNCTION: **cif_pd_track()**

PURPOSE: The generic *track* routine used by the Cognex 3-button trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_pd_track(p_pd)
   cif_pd *p_pd;
```

NOTES: *p_pd:* points to the **cif_pd** to use.

FUNCTION: **cif_realtime()**

PURPOSE: Put the frame grabber into display mode and optionally outline a window.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

GLOBALS: *cif_mode*

USE:

```
void cif_realtime(flag,p_pd)
   int flag;
   cif_pd *p_pd;
```

NOTES: *flag:* indicates whether or not to draw an outline around the window pointed to by *p_pd*'s function instance variable. An outline is drawn if *flag* is nonzero.

p_pd: points to the **cif_pd** whose *function_iv->imbuf* window is to be (optionally) outlined.

cif_realtime() calls **caq_constant()** to erase any contents of the frame grabber, optionally draws the outline of the window, then calls **caq_graphics()** to display the outline on a real-time image.

cif_realtime() cannot be used as a *start* routine by the default *track* routine because its arguments are incompatible with those passed by the *track* routine.

cif_realtime() throws *CMEM_ERR_BUS* if *p_pd* is NULL.

Behavior is unspecified if *flag* is nonzero and *p_pd->function_iv* does not refer to an initialized **cif_iv_window** instance variable.

FUNCTION: **cif_set_arms()**

PURPOSE: Sets the length of the arms for a crosshair cursor.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_set_arms(x_length,y_length,p_pd)
   int x_length, y_length;
   cif_pd *p_pd;
```

NOTES: *x_length:* the radius of the horizontal arm (in pixels).
p_pd->function_iv->x_arm is set to *x_length.*

y_length: the radius of the cursor vertical arm (in pixels).
p_pd->function_iv->y_arm is set to *y_length.*

p_pd: points to the **cif_pd** whose cursor arms are set.

cif_set_arms() throws *CGEN_ERR_BADARG* if *x_length* or *y_length* is out of range, or if *p_pd* is NULL.

Behavior is unspecified if *p_pd->function_iv* does not refer to an initialized **cif_iv_cursor** instance variable; that is, if an attempt is made to modify a cursor instance variable that does not exist.

FUNCTION: **cif_set_color()**

PURPOSE: Changes the color of a cursor, a window outline, or both.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_set_color(color,p_cursor_pd,p_window_pd)
   int color;
   cif_pd *p_cursor_pd, *p_window_pd;
```

NOTES: *color:* the new color.

p_cursor_pd: an optional pointer to a cursor **cif_pd**. If a pointer to a cursor **cif_pd** is passed, the cursor is changed to *color*. If *p_cursor_pd* is NULL, no cursor is changed. Behavior is unspecified if this points to a **cif_pd** other than a cursor **cif_pd**.

p_window_pd: an optional pointer to a window **cif_pd**. If a pointer to a window **cif_pd** is passed, the window outline is changed to *color*. If *p_window_pd* is NULL, no window outline is changed. Behavior is unspecified if this points to a **cif_pd** other than a window **cif_pd**.

FUNCTION: **cif_set_cursor()**

PURPOSE: Repositions the cursor for the specified **cif_pd**.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_set_cursor(x_coord,y_coord,p_pd)
   int x_coord,y_coord;
   cif_pd *p_pd;
```

NOTES: *x_coord:* the *x* coordinate to which to move the cursor.

y_coord: the *y* coordinate to which to move the cursor.

p_pd: points to the **cif_pd** of the cursor to be repositioned. If the new cursor coordinates are valid, *p_pd->x* is set to *x_coord* and *p_pd->y* is set to *y_coord.*

cif_set_cursor() throws *CGEN_ERR_BADARG* if:

- *x_coord* is beyond the bounds of *p_pd->xlo, p_pd->xhi;*
- *y_coord* is beyond the bounds of *p_pd->ylo, p_pd->yhi;*
- *p_pd* is NULL.

Behavior is unspecified if *p_pd->function_iv* does not refer to an initialized **cif_iv_cursor** instance variable.

FUNCTION: **cif_set_window()**

PURPOSE: Sets the size and position of a window.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_set_window(x0,y0,width,height,p_pd)
   int x0, y0;
   int width, height;
   cif_pd *p_pd;
```

NOTES: *x0:* the x coordinate of the upper left corner of the window. Specify -1 to center the window in x dimension.

y0: the y coordinate of the upper left corner of the window. Specify -1 to center the window in y dimension.

width: the width of the window. The minimum width value is 2.

height: the height of the window. The minimum height value is 2.

The window must fit within the bounds of *p_pd->display*. If *p_pd->display* refers to the frame grabber, the *x0* coordinate plus the *width* of the window must be less than *CAQ_WIDTH*, and the *y0* coordinate plus the *height* of the window must be less than *CAQ_HEIGHT*.

cif_set_window() throws the following errors:

- *CGEN_ERR_BADARG* if the window exceeds any of the bounds
- *CGEN_ERR_BADARG* if *p_pd* is NULL
- *CGEN_ERR_BADARG* if:
  - *x* is set to -1 (to center the window in *x* dimension) and *y* is set to a specific value
  - *y* is set to -1 (to center the window in *y* dimension) and *x* is set to a specific value

Behavior is unspecified if *p_pd->function_iv* does not refer to an initialized **cif_iv_window** instance variable.

FUNCTION: **cif_split_start()**

PURPOSE: Initialization routine for a video-splitting trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_split_start(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x* and *y:* currently unused; they are passed so that all *start* functions maintain the same calling convention.

p_pd: points to the split **cif_pd**. **cif_split_start()** copies *p_pd->caq_params->split* to *p_pd->y*.

cif_split_start() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_split_stop()**

PURPOSE: Reports the final screen split line before exiting the split *track* routine.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_split_stop(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* currently unused; it is passed so that all *stop* functions maintain the same calling convention.

y: the final split line.

p_pd: points to the **cif_pd** structure from which the final split line is being reported.

cif_split_stop() prints the following upon completion:

```
split at line <y>
```

FUNCTION: **cif_split_update()**

PURPOSE: Updates the screen split.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_split_update(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* currently unused; it is passed so that all *update* functions maintain the same calling convention.

y: the new screen split line.

p_pd: points to the split **cif_pd** to update. **cif_split_update()** sets *p_pd->caq_params->split* to *y*.

cif_split_update() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_std_lbut()**

PURPOSE: Used in conjunction with trackball figure manipulation to exit the trackball manipulation procedures.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

USE:

```
int cif_std_lbut(fig)
   cgr_figure *fig;
```

NOTES: **cif_std_lbut()** is normally called in response to the press of the left button on a 3-button trackball device. **cif_std_lbut()** sets the update mode for the figure to -1, thereby causing the **cif_track_figure()** function to terminate.

fig: points to the figure to which trackball manipulation applies.

FUNCTION: **cif_std_mbut()**

PURPOSE: Used in conjunction with trackball figure manipulation to increment the update mode for a figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

USE:

```
int cif_std_mbut(fig)
   cgr_figure *fig;
```

NOTES: **cif_std_mbut()** is normally called in response to the press of the middle button on a 3-button trackball device. **cif_std_mbut()** increments a figure's update mode by 1.

fig: points to the figure to which trackball manipulation applies.

FUNCTION: **cif_std_rbut()**

PURPOSE: Used in conjunction with trackball figure manipulation to decrement the update mode for a figure.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

USE:

```
int cif_std_rbut(fig)
   cgr_figure *fig;
```

NOTES: **cif_std_rbut()** is normally called in response to the press of the right button on a 3-button trackball device. **cif_std_rbut()** decrements a figure's update mode by 1.

fig: points to the figure to which trackball manipulation applies.

VARIABLE: **cif_tb_file**

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

DEFINITION:

```
FILE *cif_tb_file;
```

NOTES: *cif_tb_file* is the stream or file pointer associated with the serial line used for communication with the Cognex 3-button trackball. The functions **cif_tb3_flush()** and **cif_tb3_read()** read data from this file.

When you **#include** <*cif.v*>, it calls **cif_open_tb()** to open */dev/ser2* for trackball communication and records the returned file pointer in *cif_tb_file*.

If serial line 2 is to be used for another purpose, call **fclose()** on the file whose pointer is stored in *cif_tb_file*. To reopen it or another line for trackball communication, call **cif_open_tb()** and assign the file pointer it returns to *cif_tb_file*. Although **cif_open_tb()** opens a file, it is up to you to store the returned *FILE* pointer in *cif_tb_file*. For example:

```
cif_tb_file = cif_open_tb(devname);
```

FUNCTION: **cif_tb3_flush()**

PURPOSE: Flush a serial line prior to trackball use.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

GLOBALS: *cif_tb_file*

USE:
```
void cif_tb3_flush()
```

NOTES: Flushes the contents of the serial line associated with the global trackball pointer *cif_tb_file*, which is the device used for trackball I/O. Because trackball reads are often made after long computational delays, the standard tracking routine flushes stale data from the input buffer before each read.

cif_tb3_flush() throws *CIO_ERR_FILE* if *cif_tb_file* is NULL.

FUNCTION: **cif_tb3_read()**

PURPOSE: Read serial line data from the Cognex 3-button trackball and wait if no data is available.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

GLOBALS: *cif_tb_file*

USE:

```
void cif_tb3_read(p_x,p_y,p_buttons)
   short *p_x, *p_y, *p_buttons;
```

NOTES: *p_x:* points to the x coordinate.

p_y: points to the *y* coordinate.

p_buttons: points to the button data.

Reads the trackball position data from the device specified by *cif_tb_file*.

cif_tb3_read() throws *CIO_ERR_FILE* if *cif_tb_file* is NULL.

FUNCTION: **cif_tb3_readnw()**

PURPOSE: Read serial line data from the Cognex 3-button trackball and return immediately if no trackball data is available.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

GLOBALS: *cif_tb_file*

USE:

```
bool cif_tb3_readnw(p_x,p_y,p_buttons)
   short *p_x, *p_y, *p_buttons;
```

NOTES: *p_x:* points to the x coordinate. The x coordinate is set to zero if no trackball data was available.

p_y: points to the *y* coordinate The y coordinate is set to zero if no trackball data was available.

p_buttons: points to the button data. The button data is set to zero if no trackball data was available.

Reads the trackball position data from the device specified by *cif_tb_file*.

FUNCTION: **cif_track_figure()**

PURPOSE: Provides an interface for figures with the Cognex standard 3-button trackball.

REFERENCE: Image Processing: Advanced Graphics

INCLUDE FILE: *cgr_fig.h*

USE:

```
void cif_track_figure(fig,image)
   cgr_figure *fig;
   cip_buffer *image;
```

NOTES: **cif_track_figure()** can be used in conjunction with a figure drawn using *CGR_PUSH* draw mode, or with a figure drawn using *CGR_SET* draw mode. A trackball serial line must be opened prior to using **cif_track_figure()**.

fig: points to the figure that the trackball is to manipulate.

image: points to the image in which the figure is displayed.

FUNCTION: **cif_window_check()**

PURPOSE: Checks either the *(x,y)* coordinates or the width and height of a window for validity, depending on the trackball mode.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_window_check(p_x,p_y,p_pd)
   int *p_x, *p_y;
   cif_pd *p_pd;
```

NOTES: *p_x:* points to the *x* value being checked. If the trackball is in position mode, the *x* coordinate is checked. If the trackball is in size mode, the window width is checked.

p_pd->xlo is the lower limit for the x coordinate; *CAQ_WIDTH* minus the width of the window is the upper limit. The minimum width is two, the maximum width is *p_pd->xhi* minus the x coordinate. If the *x* value is outside of its range, *p_x* is set to the appropriate limit.

p_y: points to the *y* value being checked. If the trackball is in position mode, the y position is checked. If the trackball is in size mode, the height is checked.

p_pd->ylo is the lower limit for the y coordinate; *CAQ_HEIGHT* minus the height of the window is the upper limit. The minimum height is two, the maximum height is *p_pd->yhi* minus the y coordinate. If the *y* value is outside of its range, *p_y* is set to the appropriate limit.

p_pd: points to the window **cif_pd** to check.

cif_window_check() throws *CGEN_ERR_BADARG* if *p_px*, *p_py*, or *p_pd* is NULL.

FUNCTION: **cif_window_start()**

PURPOSE: Initialization routine for a window trackball.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_window_start(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x* and *y:* currently unused; they are passed so that all *start* functions maintain the same calling convention.

p_pd: points to the window to initialize.

cif_window_start() determines the location of the window's origin by calling **cip_overlap()** on *p_pd->display* and *p_pd->function_iv->imbuf.* It then sets *p_pd->x* to the horizontal coordinate and *p_pd->y* to the vertical coordinate. **cif_window_start()** calls **cif_realtime()** to display the outlined window.

cif_window_start() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_window_stop()**

PURPOSE: Reports the final window position, size, and area before exiting the window *track* routine.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_window_stop(x,y,p_pd)
    int x,y;
    cif_pd *p_pd;
```

NOTES: *x* and *y*: currently unused; they are passed so that all *stop* functions maintain the same calling convention.

p_pd: points to the **cif_pd** structure whose data is being used.

cif_window_stop() determines the final (x,y) coordinates of the window by calling **cip_overlap()** on *p_pd->display* and *p_pd->function_iv->imbuf*. It then sets the *x* and *y* components of *p_pd* to those values, and prints:

```
window <width> wide by <height> tall at (<x>,<y>);
area = <width*height>.
```

cif_window_stop() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_window_track()**

PURPOSE: Dynamically changes the size, position, or both, of a window.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_window_track(p_pd)
   cif_pd *p_pd;
```

NOTES: *p_pd:* points to the window **cif_pd** to change.

cif_window_track() first calls *p_pd->start* if it exists. It then reads trackball data, calls *p_pd->check* to insure the validity of the data, then calls the *p_pd->update* routine to modify the window position or size and to display the new window. **cif_window_track()** continues to execute this sequence of read, check, and update operations until the left button is pressed, at which point it calls the *p_pd->stop* routine, if it exists, to exit.

When **cif_window_track()** is invoked, the trackball begins in *position mode*. Spinning the trackball moves the window around the screen. The window is not allowed to move off any edge of the image. Pressing the middle button toggles between *position* and *size* modes. In *size* mode, the trackball shrinks or expands the right side and the bottom of the window, while the window origin remains fixed. You can switch modes as often as needed.

The right button is unused.

The low-order three bits of *p_pd->dev->buttons* reflect the state of the three trackball buttons.

cif_window_track() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

FUNCTION: **cif_window_update()**

PURPOSE: Updates the window on the screen.

REFERENCE: System Software: Interface

INCLUDE FILE: *cif_def.h*

USE:

```
void cif_window_update(x,y,p_pd)
   int x, y;
   cif_pd *p_pd;
```

NOTES: *x:* the window *x* coordinate or width. If *p_pd->function_iv->mode* is set to one, *x* is the *x* position. If *p_pd->function_iv->mode* is zero, x is the width.

y: the window *y* coordinate or height. If *p_pd->function_iv->mode* is set to one, *y* is the *y* position. If *p_pd->function_iv->mode* is zero, *y* is the height.

p_pd: points to the window to update.

If the trackball is in *position* mode the following steps occur:

- the old window is erased
- **cip_window()** is called to change the window position to reflect the new coordinates
- the resulting window is drawn

If the trackball is in *size* mode the following steps occur:

- The old window is erased.
- *p_pd->x* and *p_pd->function_iv->imbuf->width* are set to *x*.
- *p_pd->y* and *p_pd->function_iv->imbuf->height* are set to *y*.
- The resulting window is drawn.
- The frame grabber is left in real-time image mode.

cif_window_update() throws *CGEN_ERR_BADARG* if *p_pd* is NULL.

Your Comments

Please use this card to comment on this manual. In particular, please comment on the following:

Accuracy:

Completeness:

Organization:

Did you find specific errors in this manual? If you did, please list the error with its page number.

Please list any suggestions you have for future improvements.

Name_____
Title_____
Organization_____
Street_____
City_____
State_____ **Zip**_____
Date_____

Manual_____
Revision_____

COGNEX
Vision For Industry

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 1867 BOSTON, MA

POSTAGE WILL BE PAID BY:

COGNEX CORPORATION

15 Crawford Street
Needham, Massachusetts 02194

Attn: Technical Documentation

MP MASTER
MA32854